Mechanized Abstract Semantics of AADL as implemented by HAMR

Stefan Hallerstede and John Hatcliff

May 15, 2024

Contents

Part I

Static Model Structure

Chapter 1

Representing AADL Model Information

This chapter provides definitions for representing AADL static model information and associated model well-formedness specifications.

The static model structure is designed to represent HAMR-relevant content from a AADL system instance model [**?**]. Given a system model in the AADL sublanguage processed by HAMR, HAMR can generate instances of the types defined in this theory. This provides the basis for proving properties about system's structure and behavior.

(ToDo: Describe the distinction between the instance model and the information presented here.)

The theory uses Isabelle's Set and Map theories to represent model structures, but includes some additional helper functions in the SetsAndMaps theory.

theory *Model*
  imports *Main SetsAndMaps*
begin

1.1 Identifiers

This section includes types for representing different types of identifiers.

1.1.1 Port Identifiers *PortId*

Port is the main category of feature appearing on the interface of AADL software components. AADL tools typicially will need some concise way of uniquely refering to ports. When generating a model representation from AADL source, HAMR will generate a unique natural number identifier for each port, which is used throughout the HAMR run-time infrastructure. (ToDo: add remark about new representation using Slang range types.

datatype *PortId* = *PortId nat*

Sets of port identifiers are frequently used in the model representation, so we introduce an appropriate type synonym.

type-synonym *PortIds* = *PortId set*

1.1.2 Component Identifiers *CompId*

Component identifier definitions are similar to those for port identifiers. **Note:** In the current AADL formalization, threads are the only category of components represented.

datatype *CompId* = *CompId nat*

type-synonym *CompIds* = *CompId set*

1.1.3 Variable Identifiers *CompId*

Variable identifiers are also similar to port identifiers, except that strings are used for readability instead of numbers.

(ToDo: Rename the following to VarId and VarIds to match the other ID domains.)

datatype *Var* = *Var string*

type-synonym *Vars* = *Var set*

1.2 Ports Descriptors

Port descriptors combine pieces of information from the AADL instance model into a single structure that provides attributes of a component port. The descriptor includes the port's direction (in or out), kind (Event, Data, or Event Data), size (i.e., the size of the buffer associated with the port) and other user-specified properties of the port that are recognized by HAMR.

PortDirection indicates the directionality of the port. Note that HAMR only accepts unidirectional ports. AADL's bi-directional "in out" ports are disallowed because they complicate analysis, semantics, and code generation.

datatype *PortDirection* =
In | *Out*

PortKind indicates the possible AADL port category. *Event* ports model interrupt signals or other notification-oriented messages without payloads. *Data* ports model shared memory between components or distributed memory services where an update to a distributed memory cell is automatically propagated to other components that declare access to the cell. *Event data* ports model asynchronous messages with payloads, such as in publish-subscribe frameworks). Definitions in XXX cross-reference PortState.thy XXX specify the state representation for storage associated with ports. Inputs to event and event data ports are buffered. The buffer sizes and overflow policies can be configured per port using standardized AADL properties. Inputs to data ports are not buffered; newly arriving data overwrites the previous value.

datatype *PortKind* =

Event | *Data* | *EventData*

The *PortDescr* includes the following fields

- name - the printable name of the port for reporting and debugging purposes,
- id - the unique identifier for the port, as generated by the HAMR code generator,
- compId - the unique identifier for the component to which this port belongs,
- direction - the direction of the port (in or out)
- kind - the AADL port category for the port (event, data, or eventdata),
- size - the size of the buffer associated with the port, as declared by the Queue_Size port property in the AADL model (see the AADL standard Section 8.3). When a size is not specified in the AADL model, the size value defaults to 1). Data ports always have a size of 1.

record *PortDescr* =
  name :: *string*
  id :: *PortId*
  compId :: *CompId*
  direction :: *PortDirection*
  kind :: *PortKind*
  size :: *nat*
  urgency :: *nat*

1.2.1 Port Descriptor Well-formedness

A port descriptor is well-formed iff

- the queue size specified in the port descriptor is greater than 0.
- if the port is a data port, then its queue size must be equal to 1 (as specified in the AADL standard Section 8.3 (3)).

definition *wf-PortDescr* :: *PortDescr* ⇒ *bool*
  where *wf-PortDescr pd* ≡ (*PortDescr.size pd* > *0*)
                  ∧ (*PortDescr.kind pd* = *Data* ⟶ *PortDescr.size pd* = *1*)

1.2.2 Helper Functions for Working with Ports and Port Descriptors

The following function can be used to abbreviate the declaration of port descriptors.

fun *mkPortDescr* **where** *mkPortDescr n i ci d k s u*
  = (| *name*= *n*, *id*=*i*, *compId*= *ci*, *direction*=*d*, *kind*= *k*, *size*= *s*, *urgency*= *u* |)

The following helper functions query properties of ports as captured in port descriptors.

Is the port an input port?

fun *isInPD* :: *PortDescr* ⇒ *bool* **where**
isInPD pd = (*direction pd* = *In*)

Is the port an output port?

fun *isOutPD* :: *PortDescr* ⇒ *bool* **where**
isOutPD pd = (*direction pd* = *Out*)

Is the port a data port?

fun *isDataPD* :: *PortDescr* ⇒ *bool* **where**
isDataPD pd = (*kind pd* = *Data*)

Is the port an event port?

fun *isEventPD* :: *PortDescr* ⇒ *bool* **where**
isEventPD pd = (*kind pd* = *Event*)

Is the port an event data port?

fun *isEventDataPD* :: *PortDescr* ⇒ *bool* **where**
isEventDataPD pd = (*kind pd* = *EventData*)

Is the port an event-like port? Note: we use the term *event-like* to refer to ports that are either event ports or event data ports. This combined reference is useful because event-like ports are queued and have similar port update semantics, whereas data ports are not queued.

fun *isEventLikePD* :: *PortDescr* ⇒ *bool* **where**
isEventLikePD pd = ((*kind pd* = *Event*) ∨ (*kind pd* = *EventData*))

1.3 Component Descriptors

Similar to port descriptors, Component Descriptors combine pieces of information from the AADL instance model into a single structure that provides attributes of a component. In the current formalization, only Thread components are represented. Therefore, properties included in the component descriptor pertain to AADL Thread components.

The following data type represents possible values of the AADL Dispatch_Protocol Thread property (See the AADL standard Section 5.4.2 (45). Periodic threads are dispatched at regular intervals as defined by the AADL Period property (time and associated timing properties are not represented currently). Sporadic threads are dispatched up the arrival of messages on event-like ports. The specific conditions for thread dispatching are formalized in DispatchLogic.thy XXXX cross reference XXXX. HAMR currently does not support the remaining AADL dispatch protocols (Aperiodic, Timed, Hybrid).

datatype *DispatchProtocol* =
Periodic | *Sporadic*

The *CompDescr* includes the following fields

- name - the printable name of the component for reporting and debugging purposes,
- id - the unique identifier for the component, as generated by the HAMR code generator,

- portIds - set of unique identifiers for the ports that are declared on the interface of the component, for the component to which this port belongs,
- dispatchProtocol - the value of the AADL property Dispatch_Protocol for this (thread) component,
- dispatchTriggers - the set of identifiers for event-like input ports that can act as dispatch triggers for this thread (for a longer discussion, see XXXX DispatchLogic.thy XXX),
- compVars - the set of identifiers for variables that contribute to the behavior of the component, as specified by GUMBO contract declarations.

record *CompDescr* =
name :: *string*
id :: *CompId*
portIds :: *PortIds*
dispatchProtocol :: *DispatchProtocol*
dispatchTriggers :: *PortIds*
compVars :: *Vars*

1.3.1 Helper Functions for Working with Components and Component Descriptors

The following function can be used to abbreviate the declaration of component descriptors.

fun *mkCompDescr* **where** *mkCompDescr n i pis dp dts v* =
(| *name*= *n*, *id*=*i*, *portIds*= *pis*, *dispatchProtocol*= *dp*, *dispatchTriggers*= *dts*, *compVars*= *v* |)

The following helper functions query properties of components as captured in component descriptors.

Is the component a periodic thread?

fun *isPeriodicCD* :: *CompDescr* ⇒ *bool* **where**
isPeriodicCD cd = (*dispatchProtocol cd* = *Periodic*)

Is the component a sporadic thread?

fun *isSporadicCD* :: *CompDescr* ⇒ *bool* **where**
isSporadicCD cd = (*dispatchProtocol cd* = *Sporadic*)

1.4 Connections

Connections are represented as a map from a connection source *PortId* to a set of one or more target *PortIds*.

type-synonym *Conns* = (*PortId*, *PortIds*) *map*

1.5 Models

The complete static model consists of three maps (lookup tables):

- modelCompDescrs: associates component ids to component descriptors,
- modelPortDescrs: associates port ids to port descriptors,
- modelConns: associates each source port id to a set of target port ids to which it is connected.

record *Model* =
  modelCompDescrs :: (*CompId*, *CompDescr*) *map*
  modelPortDescrs :: (*PortId*, *PortDescr*) *map*
  modelConns :: *Conns*

A helper function for abbreviating the construction of model structures.

(I'm pretty sure I saw a built-in notation for this in the Isabelle tutorial (i.e., check to see if we can use the built in notation and avoid making our own everywhere.)

fun *mkModel* **where** *mkModel compdescrs portdescrs conns* =
  (| *modelCompDescrs*= *compdescrs*, *modelPortDescrs*= *portdescrs*,
    modelConns= *conns* |)

1.6 Model Helper Functions

This section defines helper function for accessing model elements.

1.6.1 Model-wide Queries About Components and Ports

The first set of helper functions are queries across an entire model (not limited to a particular component).

Return the component identifiers in model m.

fun *modelCIDs*:: *Model* ⇒ *CompId set*
  where *modelCIDs m* = *dom* (*modelCompDescrs m*)

Return the port identifiers in model m.

fun *modelPIDs*:: *Model* ⇒ *PortId set*
  where *modelPIDs m* = *dom* (*modelPortDescrs m*)

Does model m include a component (id) c?

fun *inModelCID* :: *Model* ⇒ *CompId* ⇒ *bool*
  where *inModelCID m c* = (*c* ∈ *modelCIDs m*)

Does model m include a port (id) p?

fun *inModelPID* :: *Model* ⇒ *PortId* ⇒ *bool*
  where *inModelPID m p* = (*p* ∈ *modelPIDs m*)

Does model m include a input port (id) p?

fun *isInPID* :: *Model* ⇒ *PortId* ⇒ *bool*
  where *isInPID m p* = (*direction* (*modelPortDescrs m* $ *p*) = *In*)

Does model m include an output port (id) p?

fun *isOutPID* :: *Model* ⇒ *PortId* ⇒ *bool*
where *isOutPID m p* = (*direction* (*modelPortDescrs m* $ *p*) = *Out*)

Does model m include a port (id) p with queue capacity n?

fun *isSizePID* :: *Model* ⇒ *PortId* ⇒ *nat* ⇒ *bool*
where *isSizePID m p n* = (*size* (*modelPortDescrs m* $ *p*) = *n*)

Return the queue capacity of port (id) p in model m.

fun *sizePID* :: *Model* ⇒ *PortId* ⇒ *nat*
where *sizePID m p* = (*size* (*modelPortDescrs m* $ *p*))

Return the kind (data, event, event data) of port (id) p in model m.

fun *kindPID* :: *Model* ⇒ *PortId* ⇒ *PortKind*
where *kindPID m p* = (*kind* (*modelPortDescrs m* $ *p*))

Does model m include a data port (id) p?

fun *isDataPID* :: *Model* ⇒ *PortId* ⇒ *bool*
where *isDataPID m p* = (*kindPID m p* = *Data*)

Does model m include an event port (id) p?

fun *isEventPID* :: *Model* ⇒ *PortId* ⇒ *bool*
where *isEventPID m p* = (*kindPID m p* = *Event*)

Does model m include an event data port (id) p?

fun *isEventDataPID* :: *Model* ⇒ *PortId* ⇒ *bool*
where *isEventDataPID m p* = (*kindPID m p* = *EventData*)

Does model m include an event-like port (id) p?

fun *isEventLikePID* :: *Model* ⇒ *PortId* ⇒ *bool*
where *isEventLikePID m p* = ((*kindPID m p* = *Event*) ∨ (*kindPID m p* = *EventData*))

Return the urgency of port (id) p in model m.

fun *urgencyPID* :: *Model* ⇒ *PortId* ⇒ *nat*
where *urgencyPID m p* = (*urgency* (*modelPortDescrs m* $ *p*))

1.6.2 Queries About Ports Associated With a Specific Component

The second set of helper functions support queries about properties of a particular component, which can be indicated by its id (i.e. *CompId*) or component descriptor (i.e. *CompDescr*)

In model m, does component (id) c have a data port (id) p?

fun *isPortOfCIDPID* :: *Model* ⇒ *CompId* ⇒ *PortId* ⇒ *bool*
where *isPortOfCIDPID m c p* = (*p* ∈ (*portIds* ((*modelCompDescrs m*) $ *c*)))

In model m, does component (descriptor) cd have a var (id) v?

fun *isVarOfCD* :: *Model* ⇒ *CompDescr* ⇒ *Var* ⇒ *bool*
where *isVarOfCD m cd v* = (*v* ∈ (*compVars cd*))

In model m, does component (id) c have a var (id) v?

fun *isVarOfCID* :: *Model* ⇒ *CompId* ⇒ *Var* ⇒ *bool*
where *isVarOfCID m c v* = *isVarOfCD m* ((*modelCompDescrs m*) $ *c*) *v*

In model m, does component (descriptor) cd have an input port (id) p?

fun *isInCDPID* :: *Model* ⇒ *CompDescr* ⇒ *PortId* ⇒ *bool*
where *isInCDPID m cd p* = (*p* ∈ (*portIds cd*) ∧ *isInPD* ((*modelPortDescrs m*) $ *p*))

In model m, does component (id) c have an input port (id) p?

fun *isInCIDPID* :: *Model* ⇒ *CompId* ⇒ *PortId* ⇒ *bool*
where *isInCIDPID m c* = *isInCDPID m* ((*modelCompDescrs m*) $ *c*)

In model m, does component (descriptor) cd have an output port (id) p?"

fun *isOutCDPID* :: *Model* ⇒ *CompDescr* ⇒ *PortId* ⇒ *bool*
where *isOutCDPID m cd p* = (*p* ∈ (*portIds cd*) ∧ *isOutPD* ((*modelPortDescrs m*) $ *p*))

In model m, does component (id) c have an output port (id) p?"

fun *isOutCIDPID* :: *Model* ⇒ *CompId* ⇒ *PortId* ⇒ *bool*
where *isOutCIDPID m c* = *isOutCDPID m* ((*modelCompDescrs m*) $ *c*)

In model m, does component (descriptor) cd have a data port (id) p?"

fun *isDataCDPID* :: *Model* ⇒ *CompDescr* ⇒ *PortId* ⇒ *bool*
where *isDataCDPID m cd p* = (*p* ∈ (*portIds cd*) ∧ *isDataPD* ((*modelPortDescrs m*) $ *p*))

In model m, does component (id) c have a data port (id) p?"

fun *isDataCIDPID* :: *Model* ⇒ *CompId* ⇒ *PortId* ⇒ *bool*
where *isDataCIDPID m c* = *isDataCDPID m* (*modelCompDescrs m* $ *c*)

In model m, does component (descriptor) cd have an event port (id) p?"

fun *isEventCDPID* :: *Model* ⇒ *CompDescr* ⇒ *PortId* ⇒ *bool*
where *isEventCDPID m cd p* = (*p* ∈ (*portIds cd*) ∧ *isEventPD* ((*modelPortDescrs m*) $ *p*))

In model m, does component (id) c have an event port (id) p?"

fun *isEventCIDPID* :: *Model* ⇒ *CompId* ⇒ *PortId* ⇒ *bool*
where *isEventCIDPID m c* = *isEventCDPID m* ((*modelCompDescrs m*) $ *c*)

In model m, does component (descriptor) cd have an event-like port (id) p?"

fun *isEventLikeCDPID* :: *Model* ⇒ *CompDescr* ⇒ *PortId* ⇒ *bool*
where *isEventLikeCDPID m cd p* = (*p* ∈ (*portIds cd*) ∧ *isEventLikePD* ((*modelPortDescrs m*) $ *p*))

In model m, does component (id) c have an event-like port (id) p?"

fun *isEventLikeCIDPID* :: *Model* ⇒ *CompId* ⇒ *PortId* ⇒ *bool*
where *isEventLikeCIDPID m c* = *isEventCDPID m* ((*modelCompDescrs m*) $ *c*)

In model m, does component (descriptor) cd have a input data port (id) p?"

fun *isInDataCDPID* :: *Model* ⇒ *CompDescr* ⇒ *PortId* ⇒ *bool*
where *isInDataCDPID m cd p* = (*p* ∈ (*portIds cd*)
∧ (*let pd* = ((*modelPortDescrs m*) $ *p*)
in (*isInPD pd* ∧ *isDataPD pd*)))

In model m, does component (id) c have an input data port (id) p?"

fun *isInDataCIDPID* :: *Model* ⇒ *CompId* ⇒ *PortId* ⇒ *bool*
where *isInDataCIDPID m c* = *isInDataCDPID m* ((*modelCompDescrs m*) $ *c*)

In model m, does component (descriptor) cd have an input event-like port (id) p?"

fun *isInEventLikeCDPID* :: *Model* ⇒ *CompDescr* ⇒ *PortId* ⇒ *bool*
where *isInEventLikeCDPID m cd p* = (*p* ∈ (*portIds cd*)
∧ (*let pd* = ((*modelPortDescrs m*) $ *p*)
in (*isInPD pd* ∧ *isEventLikePD pd*)))

In model m, does component (id) c have an input event-like port (id) p?"

fun *isInEventLikeCIDPID* :: *Model* ⇒ *CompId* ⇒ *PortId* ⇒ *bool*
where *isInEventLikeCIDPID m c* = *isInEventLikeCDPID m* ((*modelCompDescrs m*) $ *c*)

Return the ports belonging to component (id) c in model m.

fun *portsOfCID* :: *Model* ⇒ *CompId* ⇒ *PortId set*
where *portsOfCID m c* = *portIds* ((*modelCompDescrs m*) $ *c*)

Return the input ports belonging to component (id) c in model m.

fun *inPortsOfCID* :: *Model* ⇒ *CompId* ⇒ *PortId set*
where *inPortsOfCID m c* = {*p* . *isInCIDPID m c p*}

Return the input data ports belonging to component (id) c in model m.

fun *inDataPortsOfCID* :: *Model* ⇒ *CompId* ⇒ *PortId set*
where *inDataPortsOfCID m c* = {*p* . *isInDataCIDPID m c p*}

Return the input event-like ports belonging to component (id) c in model m.

fun *inEventLikePortsOfCID* :: *Model* ⇒ *CompId* ⇒ *PortId set*
where *inEventLikePortsOfCID m c* = {*p* . *isInEventLikeCIDPID m c p*}

Return the dispatch triggers (port ids)belonging to component (id) c in model m.

fun *dispatchTriggersOfCID* :: *Model* ⇒ *CompId* ⇒ *PortId set*
where *dispatchTriggersOfCID m c* = *dispatchTriggers* ((*modelCompDescrs m*) $ *c*)

1.7 Model Well-formedness Properties

We now define a collection of well-formedness properties for models. The notion of well-formed model (*wf-Model*) is defined as the conjunction of all of these properties.

When HAMR generates an Isabelle representation of an AADL, all of these properties are automatically proven.

Each port descriptor in the modelPortDescrs map is well-formed.

definition *wf-Model-PortDescr* :: *Model* ⇒ *bool*
where *wf-Model-PortDescr m* ≡
(∀ *p* ∈ *dom* (*modelPortDescrs m*). *wf-PortDescr* ((*modelPortDescrs m*) $ *p*))

For each entry (p:: PortId, pd:: PortDescr) in the port descriptors map, the port id in the descriptor pd matches p.

definition *wf-Model-PortDescrsIds* :: *Model* ⇒ *bool*
where *wf-Model-PortDescrsIds m* ≡
(∀ *p* ∈ *dom* (*modelPortDescrs m*). *p* = *PortDescr.id* ((*modelPortDescrs m*) $ *p*))

For each entry (c:: CompId, cd:: CompDescr) in the component descriptors map, the comp id in the descriptor cd matches c.

definition *wf-Model-CompDescrsIds* :: *Model* ⇒ *bool*
where *wf-Model-CompDescrsIds m* ≡
(∀ *c* ∈ *dom* (*modelCompDescrs m*) . *c* = *CompDescr.id* ((*modelCompDescrs m*) $ *c*))

For each entry (p:: PortId, pd:: PortDescr) in the port descriptors map, the comp id indicating the enclosing component for the port is in the domain of the component descriptors map.

definition *wf-Model-PortDescrsCompId* :: *Model* ⇒ *bool*
where *wf-Model-PortDescrsCompId m* ≡
(∀ *p* ∈ *dom* (*modelPortDescrs m*). *PortDescr.compId* ((*modelPortDescrs m*) $ *p*) ∈ *dom* (*modelCompDescrs m*))

For each entry (c:: CompId, cd:: CompDescr) in the component descriptors map, the port ids of component's contained ports are contained in the domain of the port descriptor map.

definition *wf-Model-CompDescrsContainedPortIds* :: *Model* ⇒ *bool*
where *wf-Model-CompDescrsContainedPortIds m* ≡
(∀ *c* ∈ *dom* (*modelCompDescrs m*). (*CompDescr.portIds* ((*modelCompDescrs m*) $ *c*)) ⊆ *dom* (*modelPortDescrs m*))

For each pair of component ids c, d in the model, the sets of ids of ports belonging to those components are disjoint.

definition *wf-Model-DisjointPortIds* :: *Model* ⇒ *bool*
where *wf-Model-DisjointPortIds m* ≡
(∀ *c* ∈ *dom* (*modelCompDescrs m*).
∀ *d* ∈ *dom* (*modelCompDescrs m*).
(*c* ≠ *d* ⟶ ((*CompDescr.portIds* ((*modelCompDescrs m*) $ *c*) ∩ *CompDescr.portIds* ((*modelCompDescrs m*) $ *d*)) = {})))

For each entry (p:: PortId, s:: PortId set) in the connections map, the port id p and the port ids s = p1, ..., pn are in the domain of the port descriptor map.

definition *wf-Model-ConnsPortIds* :: *Model* ⇒ *bool*
where *wf-Model-ConnsPortIds m* ≡
(∀ *p* ∈ *dom* (*modelConns m*). (*p* ∈ *dom* (*modelPortDescrs m*))
∧ ((*modelConns m*) $ *p*) ⊆ *dom* (*modelPortDescrs m*))

For each entry (p:: PortId, s:: PortId set) in the connections map, p is an output port and the ports in p' in s are input ports and the port kinds of p and p' match.

definition *wf-Model-ConnsPortCategories* :: *Model* ⇒ *bool*
where *wf-Model-ConnsPortCategories m* ≡
(∀ *p* ∈ *dom* (*modelConns m*).(*isOutPID m p*) ∧
(∀ *p'*∈ ((*modelConns m*) $ *p*).(*isInPID m p'*) ∧ (*kindPID m p* = *kindPID m p'*)))

For each entry (p:: PortId, pd:: PortDescr) in the port map, if p is an in data port, then it has a size of at most one.

definition *wf-Model-InDataPorts* :: *Model* ⇒ *bool*
where *wf-Model-InDataPorts m* ≡
(∀ *p* ∈ *dom* (*modelPortDescrs m*).
(*isInPID m p*) ∧ (*isDataPD* (*modelPortDescrs m* $ *p*))
⟶ (*sizePID m p*) ≤ *1*)

For each entry (c:: CompId, cd:: CompDescr) in the component descriptors map, if c is Sporadic, then cd's dispatchTriggers is non-empty. HAMR currently ignores dispatch trigger declarations in periodic ports. NOTE: the AADL standard does not require that dispatch triggers are declared in Sporadic threads. The standard specifies that, in the absence of dispatch trigger declarations in Sporadic threads, ALL event-like ports are treated as dispatch triggers by default. We do not include the logic for "by default". Instead, we assume that the HAMR Isabelle model generation strategy will look for any dispatch trigger declarations for the thread in the AADL model, and if no such declarations exist, the translation will explicitly insert in dispatchTriggers field in the CompDescr, a set containing the set of event-like input ports for the thread. This simplifies the logic in the Isabelle model and HAMR code-base.

definition *wf-Model-SporadicComp* :: *Model* ⇒ *bool*
where *wf-Model-SporadicComp m* ≡
(∀ *c* ∈ *dom* (*modelCompDescrs m*). (*isSporadicCD* (*modelCompDescrs m* $ *c*))
⟶ (*dispatchTriggers* (*modelCompDescrs m* $ *c*)) ≠ *empty*)

The following top-level definition for well-formed models is the conjunction of the properties above.

definition *wf-Model* :: *Model* ⇒ *bool*
where *wf-Model m* ≡
wf-Model-PortDescr m
∧ *wf-Model-PortDescrsIds m*
∧ *wf-Model-CompDescrsIds m*
∧ *wf-Model-PortDescrsCompId m*
∧ *wf-Model-CompDescrsContainedPortIds m*
∧ *wf-Model-DisjointPortIds m*
∧ *wf-Model-ConnsPortIds m*
∧ *wf-Model-ConnsPortCategories m*
∧ *wf-Model-InDataPorts m*
∧ *wf-Model-SporadicComp m*

Finiteness of models is implied by other wf conditions, e.g. *wf-SystemSchedule*, but might occasionally needed to be assumed explicitly.

definition *finite-Model* :: *Model* ⇒ *bool*
where *finite-Model m* ≡ *finite* (*dom* (*modelCompDescrs m*)) ∧ *finite* (*dom* (*modelPortDescrs m*))

1.8 Properties Derived from Well-formedness

lemma *wf-model-implies-data-ports-capacity*:
assumes *wf-m*: *wf-Model m*
and *p-in-m*: *p* ∈ *dom* (*modelPortDescrs m*)
and *p-is-dataport*: *isDataPID m p*
shows (*sizePID m p*) = *1*
proof −
from *p-is-dataport* **have** *h1*: (*kind* (*modelPortDescrs m* $ *p*)) = *Data* **by** *auto*
from *wf-m* **have** *h2*: *wf-Model-PortDescr m* **unfolding** *wf-Model-def* **by** *auto*
from *h2* **have** *h3*: *wf-PortDescr* ((*modelPortDescrs m*) $ *p*) **by** (*simp add*: *p-in-m wf-Model-PortDescr-def*)
from *h3 h1* **show** *?thesis* **by** (*simp add*: *wf-PortDescr-def*)
qed

lemma
assumes *wf-m*: *wf-Model m*
and *p-in-m*: *p* ∈ *dom* (*modelPortDescrs m*)
and *p-is-dataport*: *isDataPID m p*
shows (*sizePID m p*) = *1*
proof −
from *p-is-dataport* **have** *h1*: (*kind* (*modelPortDescrs m* $ *p*)) = *Data* **by** *auto*
from *wf-m* **have** *h3*: *wf-PortDescr* ((*modelPortDescrs m*) $ *p*)
by (*auto simp add*: *p-in-m wf-Model-def wf-Model-PortDescr-def wf-PortDescr-def*)
from *h3 h1* **show** *?thesis* **by** (*simp add*: *wf-PortDescr-def*)
qed

lemma
⟦*wf-Model m*; *isDataPID m p*⟧ ⟹ *p* ∈ *dom* (*modelPortDescrs m*)
apply (*auto simp add*: *wf-Model-def wf-Model-PortDescr-def wf-PortDescr-def*)
sorry

end

Part II

Runtime State and Behavior

Chapter 2

Representing AADL Runtime State Information

2.1 Variable States

Real-time tasks often have local state (e.g., variables declared within the source code of a thread) that are used to store values of input devices and intermediate results of task calculations. Because the AADL standard focus on architectural specifications it has no specification notation for thread-local variables. However, the BLESS AADL behavioral specification language has the ability to specify thread-local variables for threads, and the GUMBO contract language continues and extends the concept.

In GUMBO, thread entry point contracts specify real-time task behaviors in terms of pre/post-conditions that semantically are *relations* between input port values and values of thread-local variables at the time of thread dispatch to output port values and (possibly updated) values of thread-local variables at the time at which the thread completes its entry point computation.

In this Isabelle formalization, specifications of thread application logic are not "hard-wired" to GUMBO style contracts, but they are designed to be sufficient for representing GUMBO contracts. Accordingly, thread application logic behavior is defined in terms of relations as described above (see XXXX App.thy XXXX).

To support the above concepts, the representation of thread's state includes values of thread-local variables, e.g., as declared in GUMBO state declarations. When HAMR generates the Isabelle representation of threads, it will automatically generate from GUMBO state declarations a listing of thread-local variables in each threads *CompDescr*. Intuitively, each thread state includes a *VarState* field representing a "store" that maps each *CompDescr* specified local variables to values.

theory *VarState*
  imports *Main Model SetsAndMaps*
begin

A *VarState* is used to represent the state of a thread's local variables whose value persist between thread dispatches.

A *VarState* is a map, associating a var id with a value of type 'a, representing the value of the variable.

The notion of application variable type and value is not fully developed at this point, so we parameterize the *VarState* of a type *a* representing a universal value type.

type-synonym *'a VarState* = (*Var*, *'a*) *map*

Currently, we do not have any conditions for well-formedness for a VarState. Later on, we will need to add conditions, e.g., to indicate stored values match a variable's type. So leave a placeholder for well-formedness.

definition *wf-VarState-dom* :: *'a VarState* ⇒ *Var set* ⇒ *bool* **where**
wf-VarState-dom vs vars ≡ (*dom vs*) = *vars*

definition *wf-VarState* :: *'a VarState* ⇒ *Var set* ⇒ *bool* **where**
wf-VarState vs vars ≡ *wf-VarState-dom vs vars*

end

2.2 Queues

In the AADL runtime, buffered storage for event and event data ports is represented using queues. To obtain a uniform storage representation for ports (which simplifies the semantics), our semantics also represents data port storage using queues, but well-formed properties will constrain data port queues to always have one element.

AADL port queues are of bounded size – the bound is specified using the Queue_Size port property in the AADL model, and this value is stored in the port descriptor data structure (the field *size*) defined in Model.thy.

This theory defines a *Queue* data type representation for AADL queues using Isabelle lists. The data type implements AADL's different *overflow handling protocols* that indicate what the semantics should be when clients attempt to insert a value into a full queue.

theory *Queue*
imports *Main*
begin

2.2.1 Structures

Define a datatype to represent the three options for AADL's port Overflow_Handling_Protocol (OHP) property. Add an additional Unbounded option for simplicity to be used when prototyping various aspects of the semantics.

datatype *Strategy* = *DropEarliest* | *DropLatest* | *Error* | *Unbounded*

(ToDo: reconcile the terms "Strategy", "Overflow Handling Protocol")

Define a record type to represent queue values with the following fields:

- error – when the OHP is set to Error, this field is used to indicate that the queue is in an error state.
- buffer – the representation of queue storage

- capacity – [static] the maximum number of elements that the buffer (queue) can hold. The value for this field should be equal to the Queue_Size port property in the AADL model (default is 1), which is held in the *PortDescr* from Model.thy. If the OHP is unbounded, this value is ignored.
- strategy – [static] the OHP for the queue. The value for this field should be equal to the OHP port property in the AADL model (default is DropEarliest), which is held in the *PortDescr* from Model.thy

Fields marked *static* are set at the creation time for the record and do not "change" (are preserved as copies are made of the record) during system execution. An alternative design for the formalism would be to always have the Model/PortDescr available and reference the static fields directly from the model information.

record *'a Queue* =
  error :: *bool*
  buffer :: *'a list*
  capacity :: *nat*
  strategy :: *Strategy*

(ToDo: double-check the terminology in the AADL standard and reconcile the terms "capacity", "size", "max-size", etc. used in AADL standard, Model.thy, Queue.thy)

Create a queue initialised with given buffer, capacity and strategy

fun *mk-queue* :: *'a list* ⇒ *nat* ⇒ *Strategy* ⇒ *'a Queue*
  where *mk-queue b c s* = (| *error*= *False*, *buffer*= *b*, *capacity*= *c*, *strategy*= *s* |)

Create a queue initialised with an empty buffer, capacity and strategy

fun *mk-empty-queue* :: *nat* ⇒ *Strategy* ⇒ *'a Queue*
  where *mk-empty-queue c s* = (| *error*= *False*, *buffer*= [], *capacity*= *c*, *strategy*= *s* |)

The following definitions define an order on list values.

instantiation *list* :: (*equal*) *order*
begin

fun *less-eq-list* **where** *less-eq-list x y* = (∃ *z*. *x* @ *z* = *y*)
fun *less-list* **where** *less-list x y* = (∃ *z*. *x* @ *z* = *y* ∧ *z* ≠ [])

instance
proof
  fix *x y z* :: *'a list*
  show (*x* < *y*) = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*) **by** *force*
  show *x* ≤ *x* **by** *simp*
  show *x* ≤ *y* ⟹ *y* ≤ *z* ⟹ *x* ≤ *z* **by** *fastforce*
  show *x* ≤ *y* ⟹ *y* ≤ *x* ⟹ *x* = *y* **by** *fastforce*
qed

end

(ToDo Stefan: indicate ordering on lists are used in theory/proofs. Also explain what the Isabelle constructs above are, e.g., is this an instantiation of a type class?)

2.2.2 Well-formedness Definitions

A queue is well-formed if the length of the buffer conforms to the capacity value.

```
definition wf-Queue
  where wf-Queue q ≡ (0 < capacity q) ∧
                     (strategy q ≠ Unbounded ⟶ length (buffer q) ≤ capacity q)
```

2.2.3 Operations

This section defines operations on queues. Generally, the operations work on the buffer field of the record representing the queue.

Check if the queue is empty.

```
fun isEmpty :: 'a Queue ⇒ bool
  where isEmpty q = (buffer q = [])
```

Check if the queue has exactly one element.

```
fun isOneElement :: 'a Queue ⇒ bool
  where isOneElement q = (length (buffer q) = 1)
```

Return the head (first value) from the queue.

```
fun head :: 'a Queue ⇒ 'a
  where head q = hd (buffer q)
```

Take the tail of a queue.

```
fun tail :: 'a Queue ⇒ 'a Queue
  where tail q = q (| buffer:= tl (buffer q) |)
```

Enqueue a single value.

```
fun push :: 'a Queue ⇒ 'a ⇒ 'a Queue where
  push q a =
    (case strategy q of
      DropEarliest ⇒
        (if length (buffer q) < capacity q
          then q (| buffer:= buffer q @ [a] |)
          else q (| buffer:= tl (buffer q) @ [a] |))
    | DropLatest ⇒
        (if length (buffer q) < capacity q
          then q (| buffer:= buffer q @ [a] |)
          else q)
    | Error ⇒
        (if length (buffer q) < capacity q
          then q (| buffer:= buffer q @ [a] |)
          else q (| error:= True, buffer:= [] |))
    | Unbounded ⇒ q (| buffer:= buffer q @ [a] |))
```

Enqueue a list of values.

```
fun pushQueue :: 'a Queue ⇒ 'a list ⇒ 'a Queue where
  pushQueue q q' =
    (case strategy q of
      DropEarliest ⇒
        q (| buffer:= drop (length (buffer q @ q') − capacity q) (buffer q @ q') |)
    | DropLatest ⇒
        q (| buffer:= take (capacity q) (buffer q @ q') |)
    | Error ⇒
        (if length (buffer q @ q') ≤ capacity q
          then q (| buffer:= buffer q @ q' |)
          else q (| error:= True, buffer:= [] |))
    | Unbounded ⇒ q (| buffer:= buffer q @ q' |))
```

Drop the first (head-side) n values from the queue.

```
fun drop ::  nat ⇒ 'a Queue ⇒ 'a Queue
  where drop n q = q (| buffer:= List.drop n (buffer q) |)
```

Remove all values from the queue.

```
fun clear :: 'a Queue ⇒ 'a Queue
  where clear q = q(| buffer:= [] |)
```

Set the queue buffer to a specific list of values (head corresponding to the first item in the list.

```
fun setBuffer :: 'a Queue ⇒ 'a list ⇒ 'a Queue
  where setBuffer q b = q (| buffer:= b |)
```

2.2.4 Operation Properties

head Properties

```
lemma single-queue-head: buffer q = [a] ⟹ head q = a
  by simp
```

tail Properties

tail frame properties. The *tail* doesn't change the *error*, *capacity*, or *strategy* fields.

```
lemma tail-frame-error: error (tail q) = error q
  by simp

lemma tail-frame-capacity: capacity (tail q) = capacity q
  by simp

lemma tail-frame-strategy: strategy (tail q) = strategy q
  by simp
```

tail preserves well-formedness.

```
lemma tail-wf:
  assumes wf-Queue q
  shows wf-Queue (tail q)
  using assms by (auto simp add: wf-Queue-def)
```

lemma *single-queue-tail*: *buffer q* = [*a*] ⟹ *buffer* (*tail q*) = []
by *simp*

push Properties

push doesn't change the *capacity* field.

lemma *push-frame-capacity*: *capacity* (*push q a*) = *capacity q*
by (*cases* (*strategy q*); *simp*)

push doesn't change the *strategy* field.

lemma *push-frame-strategy*: *strategy* (*push q a*) = *strategy q*
by (*cases* (*strategy q*); *simp*)

Express the transformation of *push* on the buffer when the operation won't cause the capacity to be exceeded.

lemma *push-within-capacity*:
assumes *length* (*buffer q*) < *capacity q*
shows *buffer* (*push q a*) = *buffer q* @ [*a*]
using *assms* **by** (*cases* (*strategy q*); *simp*)

Express the transformation of *push* on the error flag when the operation won't cause the capacity to be exceeded.

lemma *push-no-error*:
assumes *length* (*buffer q*) < *capacity q*
shows *error* (*push q a*) = *error q*
using *assms* **by** (*cases* (*strategy q*); *simp*)

lemma *push-wf*:
assumes *wf-Queue q*
shows *wf-Queue* (*push q v*)
using *assms* **by** (*cases* (*strategy q*); *auto simp add*: *wf-Queue-def*)

drop Properties

Queue.drop frame properties. The *Queue.drop* operation doesn't change the *error*, *capacity*, or *strategy* fields.

lemma *drop-frame-error*: *error* (*drop n q*) = *error q*
by *simp*

lemma *drop-frame-capacity*: *capacity* (*drop n q*) = *capacity q*
by *simp*

lemma *drop-frame-strategy*: *strategy* (*drop n q*) = *strategy q*

by *simp*

tail preserves well-formedness.

lemma *drop-wf*:
assumes *wf-Queue q*
shows *wf-Queue* (*drop n q*)
using *assms* **by** (*auto simp add*: *wf-Queue-def*)

clear Properties

clear frame properties. The *clear* operation doesn't change the *error*, *capacity*, or *strategy* fields.

lemma *clear-frame-error*: *error* (*clear q*) = *error q*
by *simp*

lemma *clear-frame-capacity*: *capacity* (*clear q*) = *capacity q*
by *simp*

lemma *clear-frame-strategy*: *strategy* (*clear q*) = *strategy q*
by *simp*

tail preserves well-formedness.

lemma *clear-wf*:
assumes *wf-Queue q*
shows *wf-Queue* (*clear q*)
using *assms* **by** (*simp add*: *wf-Queue-def*)

setBuffer Properties

setBuffer frame properties. The *setBuffer* operation doesn't change the *error*, *capacity*, or *strategy* fields.

lemma *setBuffer-frame-error*: *error* (*setBuffer q b*) = *error q*
by *simp*

lemma *setBuffer-frame-capacity*: *capacity* (*setBuffer q b*) = *capacity q*
by *simp*

lemma *setBuffer-frame-strategy*: *strategy* (*setBuffer q b*) = *strategy q*
by *simp*

setBuffer preserves well-formedness.

lemma *setBuffer-wf*:
assumes *wf-Queue q*
and *length b* ≤ *capacity q*
shows *wf-Queue* (*setBuffer q b*)
using *assms* **by** (*simp add*: *wf-Queue-def*)

end

2.3 Port States

An AADL thread communicates with other threads over ports. Each port has some type of storage associated with it: a data port has a memory slot to hold a single value, an event data port has a queue/buffer to hold messages, and an event port has a queue/buffer to hold signals (null messages) indicating the presence of an event. To simplify the semantics, we represent the storage for every kind of port using a queue defined in Queue.thy. Queues associated with data ports will be constrained to always have size one.

The runtime needs to be able to associate a model-declared port to storage storage for the port. In HAMR, this is implemented by associating a *PortId* to a queue data structure. In this semantics mechanization, we introduce the type *PortState* – a mapping from *PortId* to *Queue* to realize that association for each thread. For simplicity, we provide separate *PortState*s for input and output ports. Further, Hatcliff-al:ISOLA22 argued that the AADL runtime semantics implies that there is a distinction between the application's view of a port's state, and the communication infrastructure's view of a port's state (see, for example, Section 8.3.1 (7) of the standard). Thus, a *ThreadState* will include four *PortState* structures:

- *iin* - infrastructure input port state (representing the infrastructure's view of input ports)
- *ain* - application input port state (representing the thread application logic's view of input ports)
- *aout* - application output port state (representing the thread application logic's view of output ports)
- *iout* - infrastructure output port state (representing the infrastructure's view of output ports)

The PortState.thy theory provides:

- the definition of a port state data structure
- definitions of well-formed port states
- operations on port states
- properties/proofs that operations preserve well-formedness

The theory depends on SetsAndMaps.thy for the map type that implements the port state, Queue.thy for storage for each port, and Model.thy to provide the basis for well-formedness (i.e., the contents of the port states are aligned with the port declarations in the model).

theory *PortState*
  imports *Main SetsAndMaps Queue Model*
begin

2.3.1 Structures

A *PortState* maps *PortId*s to queues. Intuitively, each port state applies to a particular set of *PortId*s (e.g., the input ports of a particular thread). We will use the Isabelle Map type *dom* (domain) operation to determine the set of *PortId*s that the port state supports. "Unsupported"/"Non-applicable" ports are not in the domain, while "supported" ports are always bound to a queue value.

type-synonym *'a PortState* = (*PortId*, *'a Queue*) *map*

2.3.2 Well-formedness Definitions

A *PortState* is well-formed wrt some set of PortIds if its domain is equal to the set of PortIds. This concept is used to show that common operations on port state maintain a domain that is aligned with a set of ports declared in a component (e.g., all input ports of the component).

definition *wf-PortState-dom* :: *PortId set* ⇒ *'a PortState* ⇒ *bool* **where**
wf-PortState-dom pids ps ≡ (*dom ps*) = *pids*

A *PortState* is well-formed if every *PortId* in the port state is associated with a well-formed *Queue q* (as defined in Queue.thy) and the capacity of the *q* is equal to the model-declared size of the queue as found in the model *PortDescr pd* .

definition *wf-PortState-queue* :: *Model* ⇒ *PortId* ⇒ *'a Queue* ⇒ *bool* **where**
wf-PortState-queue m p q ≡
let pd = (*modelPortDescrs m*) $ *p*
in (*wf-Queue q* ∧ *capacity q* = *size pd*)

ToDo: Add well-formedness properties that constrain the strategy fields to match what is in the port descr for the port.

definition *wf-PortState-queues* :: *Model* ⇒ *PortId set* ⇒ *'a PortState* ⇒ *bool* **where**

wf-PortState-queues m pids ps ≡
∀ *p* ∈ *pids*. *let q* = (*ps* $ *p*)
in wf-PortState-queue m p q

The following definition conjoins the well-formedness properties above.

definition *wf-PortState* :: *Model* ⇒ *PortId set* ⇒ *'a PortState* ⇒ *bool* **where**
wf-PortState m pids ps ≡ *wf-PortState-dom pids ps*
∧ *wf-PortState-queues m pids ps*

The following helper lemmas establish properties of elements (queues, buffers) that below to well-formed port states. These are used in proofs that operations on port states preserve well-formedness.

lemma *wf-PortState-implies-wf-PortState-queue*:
assumes *wf-ps*: *wf-PortState m dom-pids ps*
and *p-in-dom*: *p* ∈ *dom-pids*
shows *wf-PortState-queue m p* (*ps* $ *p*)
using *wf-ps p-in-dom*
by (*simp add*: *wf-PortState-def wf-PortState-queues-def*)

2.3.3 Operations

We define a number of helper functions for working with port states. As a naming convention, operations with "PID" in the name take a *PortId* argument as a reference to a port; operations with "PD" in the name to a *PortDescr* as a reference to a port.

Accessor Operations

Accessor operators implement queries about the aggregate port state or individual ports. These do not perform logical updates of the port state.

Does port state *ps* map port identifier *PortId* to some queue?

fun *portDefinedPID* :: *'a PortState* ⇒ *PortId* ⇒ *bool*
where *portDefinedPID ps p* = (*p* ∈ *dom ps*)

Does port state *ps* associate a non-empty queue with port identifier *PortId*? (i.e., is data available in the port storage?)

fun *dataAvailablePID* :: *'a PortState* ⇒ *PortId* ⇒ *bool*
where *dataAvailablePID ps p* = (∃ *q* . *ps p* = *Some*(*q*) ∧ ¬*isEmpty q*)

Given a port state *ps*, return the set of port ids for which there is data available (i.e., the set of port ids that are associated with non-empty queues.

fun *dataAvailablePorts* :: *'a PortState* ⇒ *PortId set*
where *dataAvailablePorts ps* = {*p* . *dataAvailablePID ps p*}

Does the port state *ps* have any queues that have data available?

fun *dataUnavailable* :: *'a PortState* ⇒ *bool*
where *dataUnavailable ps* = (∀ *p* ∈ *dom ps*. ∀ *q*. *ps p* = *Some*(*q*) ⟶ *isEmpty q*)

Does the port state *ps* have data available on all ports in the set *pids*?

fun *readyPIDs* :: *'a PortState* ⇒ *PortId set* ⇒ *bool*
where *readyPIDs ps pids* = (∀ *p* ∈ *pids*. *dataAvailablePID ps p*)

Return the first value from *p*'s queue within port state *ps*.

fun *portHeadPID* :: *'a PortState* ⇒ *PortId* ⇒ *'a*
where *portHeadPID ps p* = *head* (*ps* $ *p*)

Return the entire buffer of *p*'s queue with port state *ps*.

fun *portBufferPID* :: *'a PortState* ⇒ *PortId* ⇒ *'a list*
where *portBufferPID ps p* = *buffer* (*ps* $ *p*)

Transformer Operations

Transformer operation perform logical updates of the port states. Sections that follow will prove well-formedness preservation properties for these.

Transform the port state *ps* by replacing *p*'s queue with queue *q*.

fun *portReplacePID* :: *'a PortState* ⇒ *PortId* ⇒ *'a Queue* ⇒ *'a PortState*
where *portReplacePID* (*ps*::*'a PortState*) *p q* = *ps*(*p* ↦ *q*)

Transform the port state *ps* by replacing *p*'s buffer with queue *b*, leaving the rest of queue (the static properties and error state) unchanged.

fun *portReplaceBufferPID* :: *'a PortState* ⇒ *PortId* ⇒ *'a list* ⇒ *'a PortState*
where *portReplaceBufferPID* (*ps*::*'a PortState*) *p b* =
(*let q-pre* = (*ps* $ *p*) *in* — get the current q
let q-post = *setBuffer q-pre b in* — update the queue with a new buffer
ps(*p* ↦ *q-post*))

Transform the port state *ps* by dequeuing one value from each of the ports in the set *pids*?

```
fun portDequeuePIDs :: 'a PortState ⇒ PortId set ⇒ 'a PortState
  where portDequeuePIDs ps pids = ps ++ (λp. if p ∈ pids then Some (tail (ps $ p)) else None)
```

Transform the port state *ps* by dequeuing one value from the port *p*?

```
fun portDequeuePID :: 'a PortState ⇒ PortId ⇒ 'a PortState
  where portDequeuePID ps p = ps(p ↦ tail (ps $ p))
```

Transform the port state *ps* by enqueueing a value *v* into *p*'s queue.

```
fun portEnqueuePID :: 'a PortState ⇒ PortId ⇒ 'a ⇒ 'a PortState
  where portEnqueuePID ps p v = ps(p ↦ push (ps $ p) v)
```

Transform the port state *ps* by clearing all the queue buffers in the set of ports *pids*'s.

```
fun clearAll :: PortId set ⇒ 'a PortState ⇒ 'a PortState
  where clearAll pids ps = (λp. if p ∈ pids then Some (clear (ps $ p)) else ps p)
```

The following property provides a "sanity check" on a couple of the opeations above: enqueueing a value and then dequeueing yields an identical queue value.

```
lemma portEnqueueDequeue-empty:
  assumes avail: portDefinedPID ps p
      and capa: capacity (ps $ p) > 0
      and empty: isEmpty (ps $ p)
    shows portDequeuePID (portEnqueuePID ps p x) p = ps
proof −
  have ∀ q ∈ dom ps. portDequeuePID (portEnqueuePID ps p x) p q = ps q
  proof
    fix q
    assume q ∈ dom ps
    show portDequeuePID (portEnqueuePID ps p x) p q = ps q
    proof (cases p = q)
      case True
      obtain e b c s where h0: ps p = Some (| error= e, buffer= b, capacity= c, strategy = s |)
        by (metis Queue.cases avail domD portDefinedPID.elims(2))
      have h1: b = []
        using empty h0 by fastforce
      have h6: length [] < capacity (ps $ p)
        using capa by fastforce
      have h4: buffer (push (ps $ p) x) = [x]
        by (metis append-Nil capa empty isEmpty.elims(2) list.size(3) push-within-capacity)
      have h5: error (push (ps $ p) x) = error (ps $ p)
        by (metis capa empty isEmpty.simps list.size(3) push-no-error)
      have h7: error (ps $ p) = e
        by (simp add: h0)
      have h2: portEnqueuePID ps p x p = Some(push (ps $ p) x)
        by simp
      have h2: portEnqueuePID ps p x p = Some (| error= e, buffer= [x], capacity= c, strategy = s |)
        using h0 h1 h2 h4 h5
          map-some-val-given[of ps p (| error= e, buffer= b, capacity= c, strategy = s |)]
```

```
      by (smt (verit, ccfv-threshold) Queue.equality Queue.select-convs(1) Queue.select-convs(2)
        Queue.select-convs(3) Queue.select-convs(4) old.unit.exhaust push-frame-capacity
        push-frame-strategy)
    have h3: portDequeuePID (portEnqueuePID ps p x) p p =
          Some (| error= e, buffer= b, capacity= c, strategy = s |)
      using h1 h2 by auto
    then show ?thesis
      using h0 by force
  next
    case False
    then show ?thesis
      by simp
  qed
 qed
 thus ?thesis
 by (metis avail fun-upd-triv fun-upd-upd portDefinedPID.elims(2) portDequeuePID.elims portEnqueuePID.elims)
qed
```

2.3.4 Operation Properties

portReplacePID **operation preserves well-formedness**

If we perform *portReplacePID* for port id p that exists within port state ps, then the resulting port state has the same domain.

```
lemma portReplacePID-preserves-wf-PortState-dom:
 assumes wf-ps-dom:   wf-PortState-dom dom-pids ps
    and p-in-dom: p ∈ dom-pids
  shows wf-PortState-dom dom-pids (portReplacePID ps p q)
using wf-ps-dom p-in-dom
 by (auto simp add: wf-PortState-dom-def)
```

If we perform *portReplacePID* for port id p that exists within well-formed port state ps and the new queue is also well-formed with respect to the model, then the queues in the resulting port state all well-formed with respect to the model.

```
lemma portReplacePID-preserves-wf-PortState-queues:
 assumes wf-ps: wf-PortState m dom-pids ps
    and p-in-dom: p ∈ dom-pids
    and wf-ps-queue: wf-PortState-queue m p q
  shows wf-PortState-queues m dom-pids (portReplacePID ps p q)
 using wf-ps  — assume we start with well-formed port states
       p-in-dom
       wf-ps-queue — assume the new value of the queue is well-formed
       wf-PortState-queue-def  — well-formedness definitions and associated properties
       wf-PortState-queues-def
       wf-PortState-implies-wf-PortState-queue
 by fastforce
```

portReplacePID preserves port state well-formedness.

```
lemma portReplacePID-preserves-wf-PortState:
```

assumes *wf-ps*: *wf-PortState m dom-pids ps*
and *p-in-dom*: $p \in$ *dom-pids*
and *wf-ps-queue*: *wf-PortState-queue m p q*
shows *wf-PortState m dom-pids* (*portReplacePID ps p q*)
using *wf-ps* — assume we start with well-formed port states
p-in-dom
wf-ps-queue — assume the new value of the queue is well-formed
portReplacePID-preserves-wf-PortState-dom — previous theorems
portReplacePID-preserves-wf-PortState-queues
wf-PortState-def — primary definition
by *blast*

portReplaceBufferPID **operation preserves well-formedness**

If we perform *portReplaceBufferPID* for port id p that exists within port state *ps*, then the resulting port state has the same domain.

lemma *portReplaceBufferPID-preserves-wf-PortState-dom*:
assumes *wf-ps-dom*: *wf-PortState-dom dom-pids ps*
and *p-in-dom*: $p \in$ *dom-pids*
shows *wf-PortState-dom dom-pids* (*portReplaceBufferPID ps p b*)
using *wf-ps-dom p-in-dom*
by (*auto simp add*: *wf-PortState-dom-def*)

****Update***** If we perform *portReplaceBufferPID* for port id p that exists within well-formed port state *ps* and the new queue is also well-formed with respect to the model, then the queues in the resulting port state all well-formed with respect to the model.

lemma *portReplaceBufferPID-preserves-wf-PortState-queues*:
assumes *wf-ps*: *wf-PortState m dom-pids ps*
and *p-in-dom*: $p \in$ *dom-pids*
and *b-wf*: *length* $b \leq$ (*sizePID m p*)
shows *wf-PortState-queues m dom-pids* (*portReplaceBufferPID ps p b*)
using *wf-ps* — assume we start with well-formed port states
p-in-dom
b-wf — assume the new value of the buffer is well-formed
wf-PortState-queue-def — well-formedness definitions and associated properties
wf-PortState-queues-def
wf-PortState-implies-wf-PortState-queue
setBuffer-wf — setting wf buffer within wf queue produces wf queue
by *fastforce*

portReplacePID preserves port state well-formedness.

lemma *portReplaceBufferPID-preserves-wf-PortState*:
assumes *wf-ps*: *wf-PortState m dom-pids ps*
and *p-in-dom*: $p \in$ *dom-pids*
and *b-wf*: *length* $b \leq$ (*sizePID m p*)
shows *wf-PortState m dom-pids* (*portReplaceBufferPID ps p b*)
using *wf-ps* — assume we start with well-formed port states
p-in-dom

```
    b-wf — assume the new buffer is well-formed wrt queue capacity
    portReplaceBufferPID-preserves-wf-PortState-dom — previous theorems
    portReplaceBufferPID-preserves-wf-PortState-queues
    wf-PortState-def — primary definition
  by blast
```

portDequeuePID **operation preserves well-formedness**

If we perform *portDequeuePID* for port id p that exists within port state ps, then the resulting port state has the same domain.

```
lemma portDequeuePID-preserves-wf-PortState-dom:
 assumes wf-ps-dom:   wf-PortState-dom dom-pids ps
    and p-in-dom: p ∈ dom-pids
  shows wf-PortState-dom  dom-pids (portDequeuePID ps p)
 using wf-ps-dom p-in-dom
 by (auto simp add: wf-PortState-dom-def)
```

If we perform *portDequeuePID* for port id p that exists within port state ps, then the resulting queue is well-formed.

```
lemma portDequeuePID-preserves-wf-PortState-queue:
 assumes wf-ps: wf-PortState m dom-pids ps
    and p-in-dom: p ∈ dom-pids
  shows wf-PortState-queue m p ((portDequeuePID ps p) $ p)
proof −
 from wf-ps p-in-dom have wf-operand-queue: wf-PortState-queue m p (ps $ p)
  by (rule wf-PortState-implies-wf-PortState-queue)
 show ?thesis
  using wf-operand-queue
       tail-wf
       wf-PortState-queue-def
   by fastforce

 qed
```

All the other queues within the port state ps not operated on by *portDequeuePID* are unchanged.

```
lemma portDequeuePID-frame:
 assumes wf-ps: wf-PortState m dom-pids ps
    and p-in-dom: p ∈ dom-pids
  shows ∀ p'∈ dom-pids − {p} .  ((portDequeuePID ps p) $ p') = ps $ p'
 by simp
```

If we perform *portDequeuePID* for port id p that exists within port state ps, then all the queues in the resulting port state are well-formed.

```
lemma portDequeuePID-preserves-wf-PortState-queues:
 assumes wf-ps: wf-PortState m dom-pids ps
    and p-in-dom: p ∈ dom-pids
  shows wf-PortState-queues m dom-pids (portDequeuePID ps p)
 using wf-ps — assume we start with well-formed port states
```

wf-PortState-queues-def — ..which implies that we have well-formed queues
wf-PortState-implies-wf-PortState-queue — ..which implies that the argument to dequeue is well-formed
portDequeuePID-preserves-wf-PortState-queue — ..and dequeue produces a well-formed queue
by *fastforce*

portDequeuePID preserves port state well-formedness.

lemma *pdeq-preserves-wf-PortState*:
assumes *wf-ps*: *wf-PortState m dom-pids ps*
and *p-in-dom*: $p \in$ *dom-pids*
shows *wf-PortState m dom-pids* (*portDequeuePID ps p*)
using *wf-ps p-in-dom*
portDequeuePID-preserves-wf-PortState-dom
portDequeuePID-preserves-wf-PortState-queues
wf-PortState-def
by *blast*

portEnqueuePID operation preserves well-formedness

If we perform *portEnqueuePID* for port id p that exists within port state *ps*, then the resulting port state has the same domain.

lemma *portEnqueuePID-preserves-wf-PortState-dom*:
assumes *wf-ps-dom*: *wf-PortState-dom dom-pids ps*
and *p-in-dom*: $p \in$ *dom-pids*
shows *wf-PortState-dom dom-pids* (*portEnqueuePID ps p v*)
using *wf-ps-dom p-in-dom*
by (*auto simp add*: *wf-PortState-dom-def*)

If we perform *portEnqueuePID* for port id p that exists within port state *ps*, then the resulting queue is well-formed.

lemma *portEnqueuePID-preserves-wf-PortState-queue*:
assumes *wf-ps*: *wf-PortState m dom-pids ps*
and *p-in-dom*: $p \in$ *dom-pids*
shows *wf-PortState-queue m p* ((*portEnqueuePID ps p v*) $ *p*)

proof −
from *wf-ps p-in-dom* **have** *wf-operand-portstate-queue*: *wf-PortState-queue m p* (*ps* $ *p*)
by (*rule wf-PortState-implies-wf-PortState-queue*)
from *p-in-dom wf-operand-portstate-queue*
have *wf-result-push-queue*: *wf-Queue* (*push* (*ps* $ *p*) *v*)
using *push-wf wf-PortState-queue-def*
by *metis*
from *p-in-dom wf-result-push-queue*
have *wf-result-queue*: *wf-Queue* ((*portEnqueuePID ps p v*) $ *p*)
using *push-wf p-in-dom wf-operand-portstate-queue*
by *simp*
have *capacity-preserved*: *capacity* ((*portEnqueuePID ps p v*) $ *p*) = *capacity* (*ps* $ *p*)
using *p-in-dom push-frame-capacity*
by (*metis fun-upd-same map-some-val-given portEnqueuePID.simps*)

from *p-in-dom wf-operand-portstate-queue wf-result-queue capacity-preserved*
show *?thesis*
by (*metis wf-PortState-queue-def*)
qed

If we perform *portEnqueuePID* for port id *p* that exists within well-formed port state *ps*, then the queues in the resulting port state all well-formed with respect to the model.

lemma *portEnqueuePID-preserves-wf-PortState-queues*:
assumes *wf-ps*: *wf-PortState m dom-pids ps*
and *p-in-dom*: *p* ∈ *dom-pids*
shows *wf-PortState-queues m dom-pids* (*portEnqueuePID ps p v*)
using *wf-ps* — assume we start with well-formed port states
p-in-dom
portEnqueuePID-preserves-wf-PortState-queue
wf-PortState-def
wf-PortState-queues-def — well-formedness definitions and associated properties
by (*smt* (*verit*, *best*) *fun-upd-apply map-get-def portEnqueuePID.simps*)

portEnqueuePID preserves port state well-formedness.

lemma *portEnqueuePID-preserves-wf-PortState*:
assumes *wf-ps*: *wf-PortState m dom-pids ps*
and *p-in-dom*: *p* ∈ *dom-pids*
shows *wf-PortState m dom-pids* (*portEnqueuePID ps p v*)
using *wf-ps p-in-dom* — assumptions
portEnqueuePID-preserves-wf-PortState-dom — lemmas showing subproperties of wf preserved
portEnqueuePID-preserves-wf-PortState-queues
wf-PortState-def — definition of well-formedness
by *metis*

end

2.4 Thread States

The state of a thread includes the state of its ports, local variables, and its dispatch status (a structure indicating what caused the dispatch of the thread and what input ports are currently frozen).
A thread's state changes due to:

- execution of the thread's entry points, with the semantics of execution being reflected in the application logic (see App.thy) for each entry point. Entry point execution may change the application view of the input and output port states and local variables.
- communication actions in the communication infrastructure. This may change the infrastructure view of the input ports (e.g., as messages arrive at the thread's inputs) and output ports (e.g., as messages are released from the thread onto the communication substrate).
- execution of AADL run-time services as the thread changes its scheduling state. This includes the transfer port values from infrastructure input ports to application input ports via the ReceiveInput

RTS – "freezing" the application's view of the input ports, as well as the transfer of port values from application output ports to infrastructure output ports via the SendOutput RTS.

Other theories will prove that, regardless of state changes, the thread state will always be well-formed according to the definitions in this theory.

This theory defines:

- the structure of a thread state,
- notions of well-formedness for a thread state, and
- a characterization of the valid initial states for a thread.

The theory imports SetsAndMaps.thy to represent basic structures, Models.thy to supply variable and port identifiers and to the align the variable states and port states with model declarations to achieve well-formedness, and VarState.thy and PortState.thy to provide the representations of variable state and port state.

theory *ThreadState*
  imports *Main SetsAndMaps Model VarState PortState*
begin

2.4.1 Structures

The AADL standard presents "dispatch status" as information that the thread application code can access to find out what triggered the dispatch of the thread (in particular, for sporadic dispatch protocol, which input event-like port had an event arrival that triggered the dispatch). However, the standard does not fully specify the concept.

The following datatype defines our interpretation of the dispatch status.

datatype *DispatchStatus* =
    NotEnabled
  | *Periodic PortIds*
  | *Sporadic PortId* ∗ *PortIds*

The "Periodic" alternate of the datatype indicates that the thread is currently executing due to a periodic dispatch and the accompanying *PortIds* indicates the set of input ports that have had their values frozen (by the invocation of the ReceiveInput RTS). The "Sporadic" alternate of the datatype indicates that the thread is currently executing due to a sporadic dispatch that has been triggered by the arrival of a message on the port indicated by *PortId* and the accompanying *PortIds* indicates the set of input ports that have had their values frozen. According to the AADL standard, the thread application code should never access any port whose values are not frozen (the rationale is that such an access could result in a race condition). The condition that the thread (and, in HAMR, an accompanying GUMBO contract) must not access an unfrozen port should be enforced by static checks on the program code and contract specification. For more information about how this semantics interprets the standard's approach to "freezing input ports", see DispatchLogic.thy.

The "NotEnabled" alternate is more a technical choice of the semantics design. The thread application code would never see this alternate since if the thread is not enabled, the application code would not be running. We have added this alternative to give the runtime system a value that the dispatch status

field of the thread state can be set to when the thread is not executing. Another alterative might be to have the status field set to the most recent dispatch status value or some other default value.

Given a dispatch status value, the following helper function returns the set of input ports whose values are frozen. These form the conceptual "port inputs" to the application logic during that particular dispatch of the thread.

```
fun dispatchInputPorts :: DispatchStatus ⇒ PortIds where
  dispatchInputPorts NotEnabled = {}
| dispatchInputPorts (Periodic ps) = ps
| dispatchInputPorts (Sporadic (-, ps)) = ps
```

The following helper predicate is used in thread state well-formedness definitions. It holds when p is a port id mentioned any where in the dispatch status. Such a port id should appear in the set of input port ids for a thread.

```
fun disp-elem:: DispatchStatus ⇒ PortId ⇒ bool where
  disp-elem ds p = (case ds of
                      NotEnabled ⇒ False
                    | Periodic portset ⇒ p ∈ portset
                    | Sporadic (p',portset) ⇒ ((p = p') ∨ p ∈ portset))
```

The runtime state of the thread consists of the following elements. For further motivation and rationale, see Hatcliff-al:ISOLA22. The justification for the *PortState* fields of the *ThreadState* is also summarized in PortState.thy.

- *tvar* - the state of the thread's local variables
- *iin* - infrastructure input port state (representing the infrastructure's view of input ports)
- *ain* - application input port state (representing the thread application logic's view of input ports)
- *aout* - application output port state (representing the thread application logic's view of output ports)
- *iout* - infrastructure output port state (representing the infrastructure's view of output ports)
- *disp* - the current dispatch status of the thread

```
record 'a ThreadState =
  infi :: 'a PortState
  appi :: 'a PortState
  appo :: 'a PortState
  info :: 'a PortState
  tvar :: 'a VarState
  disp :: DispatchStatus
```

The following function helps abbreviate the construction of a thread state.

```
fun tstate where tstate ii ai ao io tv ds =
(| infi= ii, appi= ai, appo= ao, info= io, tvar= tv, disp= ds |)
```

2.4.2 Well-formedness Definitions

In general, thread state well-formedness definitions specify that the things (vars, ports) that we are manipulating in the state for a thread t are aligned with things that we declared in the model for t. (e.g., the thread state does not include a queue for a port that was not declared for the thread in the model, and conversely, every port that was declared for this thread in the model has a queue associated with it). First, well-formedness conditions for each of the thread state elements are specified. Then, the well-formedness condition for the entire thread state is defined as a conjunction of these properties.

Well-formed Thread State Elements

definition *wf-ThreadState-tvar*:: *Model* ⇒ *CompId* ⇒ (*'a VarState*) ⇒ *bool* **where**
wf-ThreadState-tvar m c vs ≡ *wf-VarState vs* {*v* . *isVarOfCID m c v*}

The infi component of a ThreadState (input infrastructure port map) is well formed when the domain of the infi port map is equal to the set of input ports for the thread declared in the model. Intuitively, each of the declared "in" ports for the thread (according to the model) is associated with a infrastructure message queue, (and there are no "extra" ports in the map).

definition *wf-ThreadState-infi*:: *Model* ⇒ *CompId* ⇒ (*'a PortState*) ⇒ *bool* **where**
wf-ThreadState-infi m c ps ≡ *wf-PortState m* {*p* . *isInCIDPID m c p*} *ps*

The definitions below for other port-state elements are similar.

definition *wf-ThreadState-appi*:: *Model* ⇒ *CompId* ⇒ (*'a PortState*) ⇒ *bool* **where**
wf-ThreadState-appi m c ps ≡ *wf-PortState m* {*p* . *isInCIDPID m c p*} *ps*

definition *wf-ThreadState-appo*:: *Model* ⇒ *CompId* ⇒ (*'a PortState*) ⇒ *bool* **where**
wf-ThreadState-appo m c ps ≡ *wf-PortState m* {*p* . *isOutCIDPID m c p*} *ps*

definition *wf-ThreadState-info*:: *Model* ⇒ *CompId* ⇒ (*'a PortState*) ⇒ *bool* **where**
wf-ThreadState-info m c ps ≡ *wf-PortState m* {*p* . *isOutCIDPID m c p*} *ps*

If p is mentioned in the dispatch status of ts, then it must be an input port of c. ToDo: constrain to dispatch triggers, also check the relationship between p' and portset in the Sporadic case.

definition *wf-ThreadState-disp*:: *Model* ⇒ *CompId* ⇒ *DispatchStatus* ⇒ *bool* **where**
wf-ThreadState-disp m c ds ≡ (∀ *p*. *disp-elem ds p* ⟶ *isInCIDPID m c p*)

Well-formed Thread States

definition *wf-ThreadState*:: *Model* ⇒ *CompId* ⇒ (*'a ThreadState*) ⇒ *bool*
where *wf-ThreadState m t ts* ≡
(*wf-ThreadState-infi m t* (*infi ts*)) ∧
(*wf-ThreadState-appi m t* (*appi ts*)) ∧
(*wf-ThreadState-appo m t* (*appo ts*)) ∧
(*wf-ThreadState-info m t* (*info ts*)) ∧
(*wf-ThreadState-tvar m t* (*tvar ts*)) ∧
(*wf-ThreadState-disp m t* (*disp ts*))

2.4.3 Initial Thread States

Characterizing system execution, e.g., in terms of traces, requires some notion of initial system state, which in turn requires some notion of initial thread state. This section provides definitions characterizing initial thread states. Following AADL's philosophy of a system Initialization phase, the Initialize entry point for each thread *should* provide the initial state of the thread that will be seen in the "normal" AADL system Compute phase by the thread's application code. However, from a technical standpoint in the semantics, we need an initial state for the Initialization phase (which is what is provided in this section). Since the code of an thread Initialize entry point should never read the initial values of thread local variables nor the initial values of ports, the semantics of the thread's application logic should not be dependent on the initial thread state. Thus, we have some freedom regarding what we choose as the initial state (particular for variables). The AADL standard implicitly assumes that all port queues start out empty. So that is reflected in the definitions below. For variables, we currently choose an arbitrary default value for variables. This implies that the current definitions will yield a single unique initial state. However, since Initialize entry point application code should initialize all local variables, it would also be appropriate in the semantics to leave the *tvar* thread state component with arbitrary values for each of the local variables declared for the thread (yielding a family of initial states for a thread).

Other aspects of the initial state predicates reference the well-formed state predicates to ensure that the domains used in variable value and port value maps match the model declarations.

Currently, we instantiate the universal data type to *int* and use $0::'a$ for the default variable value.

definition *default-value*:: *int* **where** *default-value* $\equiv$ *0*

The *tvar* component of an initial state is well-formed wrt the model, and the value of each variable is the default value.

definition *initial-ThreadState-tvar*:: *Model* $\Rightarrow$ *CompId* $\Rightarrow$ *int ThreadState* $\Rightarrow$ *bool* **where**
initial-ThreadState-tvar m c ts $\equiv$
(*wf-ThreadState-tvar m c* (*tvar ts*))
$\wedge$ ($\forall$ *v* $\in$ *dom* (*tvar ts*) . (*tvar ts*) *v* = *Some*(*default-value*))

For each port state component of the thread state, the port state should be well-formed wrt the model and should be associated with an empty queue.

definition *initial-ThreadState-infi*:: *Model* $\Rightarrow$ *CompId* $\Rightarrow$ *int ThreadState* $\Rightarrow$ *bool* **where**
initial-ThreadState-infi m c ts $\equiv$ (*wf-ThreadState-infi m c* (*infi ts*)) $\wedge$ *dataUnavailable* (*infi ts*)

definition *initial-ThreadState-appi*:: *Model* $\Rightarrow$ *CompId* $\Rightarrow$ *int ThreadState* $\Rightarrow$ *bool* **where**
initial-ThreadState-appi m c ts $\equiv$ (*wf-ThreadState-appi m c* (*appi ts*)) $\wedge$ *dataUnavailable* (*appi ts*)

definition *initial-ThreadState-appo*:: *Model* $\Rightarrow$ *CompId* $\Rightarrow$ *int ThreadState* $\Rightarrow$ *bool* **where**
initial-ThreadState-appo m c ts $\equiv$ (*wf-ThreadState-appo m c* (*appo ts*)) $\wedge$ *dataUnavailable* (*appo ts*)

definition *initial-ThreadState-info*:: *Model* $\Rightarrow$ *CompId* $\Rightarrow$ *int ThreadState* $\Rightarrow$ *bool* **where**
initial-ThreadState-info m c ts $\equiv$ (*wf-ThreadState-info m c* (*info ts*)) $\wedge$ *dataUnavailable* (*info ts*)

The initial dispatch status of the thread is *NotEnabled*.

definition *initial-ThreadState-disp*:: *Model* $\Rightarrow$ *CompId* $\Rightarrow$ *int ThreadState* $\Rightarrow$ *bool* **where**
initial-ThreadState-disp m c ts $\equiv$ (*wf-ThreadState-disp m c* (*disp ts*)) $\wedge$ (*disp ts* = *NotEnabled*)

We take the conjunction of the conditions for components of the thread state to get the overall predicate for a valid initial thread state.

definition *initial-ThreadState*:: *Model* ⇒ *CompId* ⇒ (*int ThreadState*) ⇒ *bool*
where *initial-ThreadState m t ts* ≡
(*initial-ThreadState-infi m t ts*) ∧
(*initial-ThreadState-appi m t ts*) ∧
(*initial-ThreadState-appo m t ts*) ∧
(*initial-ThreadState-info m t ts*) ∧
(*initial-ThreadState-tvar m t ts*) ∧
(*initial-ThreadState-disp m t ts*)

The following lemma states that any initial thread state is well-formed.

lemma *initial-implies-wf*:
⟦*initial-ThreadState m t ts*⟧ ⟹ *wf-ThreadState m t ts*
unfolding *initial-ThreadState-def*
initial-ThreadState-infi-def initial-ThreadState-appi-def
initial-ThreadState-appo-def initial-ThreadState-info-def
initial-ThreadState-tvar-def initial-ThreadState-disp-def
wf-ThreadState-def
by *blast*

end

2.5 System States

An AADL runtime system state includes the state of each of the threads in the system, the state of the inter-thread communication substrate, and the state of various system services associated with scheduling, etc.

This theory uses definitions ThreadState.thy (for representing the state of threads) and Model.thy (for aligning the state elements with model information).

theory *SystemState*
imports *Main Model ThreadState*
begin

2.5.1 System Phase Structures

AADL executions are separated into an *Initializing* phase and a *Computing* phase (see the standard - Section 5.4.1 Clause (21), Section 13.3 Clause (7)).

In the Initializing phase, each thread's application code Initialize Entry Point executed once. The application developer designs this code to provide an initial value to each variable in the thread's state and to put initial values on its output ports.

In the Computing phase, the Compute Entry Point application code for each thread is executed repeated, according to the thread scheduling policy.

The two phases are associated with dedicated scheduling information. See datatype *Exec* below.

2.5.2 Scheduling State Structures

From the scheduler's perspective, each thread is either

- suspended awaiting dispatch,
- dispatched and ready to be scheduled,
- running.

This is a simplication of the AADL thread scheduling states reflected in Section 5.4.1 "Thread States and Actions" and 5.4.2 "Thread Dispatching" of the standard and Figures 5 and 6. In particular, we do not consider states related to modes, activation, deactivation, nor suspension due to resource acquisition or subprogram calls.

datatype *ScheduleState* = *Waiting* | *Ready* | *Running*

type-synonym *Threadschedule* = (*ScheduleState*, *ScheduleState*) *map*

The AADL runtime is design to integrate with a scheduling infrastructure on the underlying platform. The standard does not specify a particular scheduling strategy. Our scheduling-related definitions are set up (a) as a minimal abstract representation of scheduling, with (b) the ability to refine the definition to a particular scheduling strategy.

Our abstract scheduling information indicates that the system is either initializing threads (with a list of ids of threads remaining to be initialized), or is in the Computing phase and with *CompId* indicating the thread whose compute entry point will be executed next according to the underlying (unspecified) scheduling strategy.

datatype *Exec* = *Initialize CompId list* | *Compute CompId*

Since Thread Initialize entry points do not read input ports, the ordering of the execution of Initialize entry point is immaterial (see the standard Section 13.3 Clause (8). We will subsequently prove that our semantics definitions support this independence property.

For now, the notion of system schedule is instantiated to a static cyclic schedule. *scheduleInit* provides a totally ordered thread schedule for the system's initialization phase. *scheduleFirst* indicates set the threads that may be scheduled first in the system Compute phase. For a given thread t, *ScheduleComp* defines the set of threads whose execution may follow t.

record *SystemSchedule* =
  scheduleInit :: *CompId list*
  scheduleFirst :: *CompId set*
  scheduleComp :: (*CompId*, *CompId set*) *map*

2.5.3 System State Structures

The system state includes the following elements:

- a mapping from *CompId* to the thread states,
- an (abstract) representation of the state of the communication substrate,

- a mapping from *CompId* to the scheduling state of each thread,
- the current phase of the system (contained in *Exec*,
- the thread to be executed next.

record ('*u*, '*a*) *SystemState* =
  systemThread :: (*CompId*, '*a ThreadState*) *map* — states of each thread
  systemComms :: '*u* — state of communication substrate
  systemState :: (*CompId*, *ScheduleState*) *map* — schedule state of each thread
  systemExec :: *Exec* — system state and thread to be executed next

Define an instantiation of the system state in which the communication substrate structure is defined as a set of communication packets with a source *PortId* a payload, and a target *PortId*.

type-synonym '*a CommonState* = ('*a*, (*PortId* ∗ '*a* ∗ *PortId* ∗ *nat*) *set*) *SystemState*

The following helper function uses the map *ran* (range) operation to retreive the set of thread states associated with all threads in the system state *s*.

fun *systemThreadStates* :: ('*u*, '*a*) *SystemState* ⇒ '*a ThreadState set*
  where *systemThreadStates s* = *ran* (*systemThread s*)

The following helper predicates can be used to determine the current phase of the system.

The system is in the initialization phase when the *systemPhase* field is set to *Initializing* and the execution schedule field is set to an initialization schedule.

fun *isInitializing* :: ('*a*, '*u*) *SystemState* ⇒ *bool*
  where *isInitializing s* = (∃ *cs*. *systemExec s* = *Initialize cs*)

The system is in the compute phase when the *systemPhase* field is set to *Computing* and the execution schedule field indicates that component *c* is the next to execute.

fun *isComputing* :: ('*a*, '*u*) *SystemState* ⇒ *bool*
  where *isComputing s* = (∃ *c*. *systemExec s* = *Compute c*)

lemma *init-not-compute*: *isInitializing s* ⟹ ¬*isComputing s*
  apply *simp*
  by *fastforce*

lemma *compute-not-init*: *isComputing s* ⟹ ¬*isInitializing s*
  apply *simp*
  by *fastforce*

lemma *init-init*: *systemExec s* = *Initialize cs* ⟹ *isInitializing s*
  by *simp*

lemma *compute-compute*: *systemExec s* = *Compute c* ⟹ *isComputing s*
  by *simp*

2.5.4 Well-formedness Definitions

This section define a notion of well-formed system state. This is organized in terms of well-formedness properties for each system state element.

Well-formedness Definitions for Thread States

The system state's thread state map is well-formed if each thread state in the map is well-formed and if the domain of the map equals the set of thread ids in the model.

definition *wf-SystemState-ThreadStates* :: *Model* ⇒ (′*u*, ′*a*) *SystemState* ⇒ *bool*
where *wf-SystemState-ThreadStates m ss* ≡
let threadStates = *systemThread ss in*
∀ *c* ∈ *dom threadStates* . *wf-ThreadState m c* (*threadStates* $ *c*)

definition *wf-SystemState-ThreadStates-dom* :: *Model* ⇒ (′*u*, ′*a*) *SystemState* ⇒ *bool*
where *wf-SystemState-ThreadStates-dom m ss* ≡
let threadStates = *systemThread ss in*
dom threadStates = *modelCIDs m*

***** ToDo ****** communication state.

The system state's thread schedule state map is well-formed if the domain of the map equals the set of thread ids in the model.

definition *wf-SystemState-ScheduleStates-dom* :: *Model* ⇒ (′*u*, ′*a*) *SystemState* ⇒ *bool*
where *wf-SystemState-ScheduleStates-dom m ss* ≡
let scheduleStates = *systemState ss in*
dom scheduleStates = *modelCIDs m*

definition *wf-SystemState* **where** *wf-SystemState m x* ≡ *dom* (*systemThread x*) ⊆ *modelCIDs m*

Well-formedness for *Exec* indicates that (a) when in the Initializing phase the list of thread ids yet to be initialized are found in the thread ids of the model. and (b) when in the Compute phase the id of the next thread to execute is found in the thread ids of the model.

definition *wf-Exec* :: *Model* ⇒ *Exec* ⇒ *bool*
where *wf-Exec m e* ≡
case e of
Initialize cs ⇒ *set cs* ⊆ (*modelCIDs m*)
| *Compute c* ⇒ *c* ∈ (*modelCIDs m*)

Well-formedness of the system state is the conjunction of the well-formed properties above.

definition *wf-SystemStateJohn* :: *Model* ⇒ (′*u*, ′*a*) *SystemState* ⇒ *bool*
where *wf-SystemStateJohn m ss* ≡
wf-SystemState-ThreadStates m ss

$\wedge$ *wf-SystemState-ThreadStates-dom m ss*
$\wedge$ *wf-SystemState-ScheduleStates-dom m ss*
$\wedge$ *wf-Exec m* (*systemExec ss*)

The following definition gives well-formed conditions for system schedules:

- the thread ids referenced in the initialization phase schedule, must match exactly the set of thread ids in the model (i.e., those for which a model descriptor is declared), and
- the length of the initialization phase schedule must be equal to the number of threads (along with the first condition, this implies that every thread appears exactly once in the initialization phase schedule,
- there must be at least one thread given in the *scheduleFirst* set for the computing phase scheduling,
- the thread ids in *scheduleFirst* set are "valid" (i.e., they appear in the model declarations), and
- each thread id declared in the model is accounted for in the *scheduleComp* "next to schedule" map (i.e., the map is total on the model-declared threads) and every model-declared thread has an entry in the map.

definition *wf-SystemSchedule* :: *Model* $\Rightarrow$ *SystemSchedule* $\Rightarrow$ *bool*
where *wf-SystemSchedule md sc* $\equiv$
(*set* (*scheduleInit sc*) = *dom* (*modelCompDescrs md*)) $\wedge$
(*length* (*scheduleInit sc*) = *card* (*dom* (*modelCompDescrs md*))) $\wedge$
(*scheduleFirst sc* $\neq$ {}) $\wedge$
(*scheduleFirst sc* $\subseteq$ *dom* (*modelCompDescrs md*)) $\wedge$
(*dom* (*scheduleComp sc*) = *dom* (*modelCompDescrs md*))

2.5.5 Communication

record ('*u*,'*a*) *Communication* =
comPush :: '*u* $\Rightarrow$ '*a PortState* $\Rightarrow$ *Conns* $\Rightarrow$ ('*u* $\times$ '*a PortState*) *set*
comPull :: '*u* $\Rightarrow$ '*a PortState* $\Rightarrow$ *Conns* $\Rightarrow$ ('*u* $\times$ '*a PortState*) *set*

fun *commonPushItems* **where**
commonPushItems - - [] - = {}
| *commonPushItems p q* (*x*#*xs*) *mx* = {(*p*, *x*, *q*, *Suc mx*)} $\cup$ *commonPushItems p q xs* (*Suc mx*)

fun *commonPushSubstrate* **where**
commonPushSubstrate cn ps pids mx pf = ($\bigcup p \in pids.\ \bigcup q \in cn\ \$\ p.$ (*commonPushItems p q* (*pf p*) *mx*))

fun *commonPushQueues* **where**
commonPushQueues ps pf p =
(*if p* $\in$ *dom ps*
then Some (*drop* (*length* (*pf p*)) (*ps* $ *p*))
else None)

fun *commonPush* **where**
commonPush sb ps cn =

```
  {(tb, qs) | tb qs pids pf.
    pids ⊆ dom ps ∧
    (∀ p ∈ pids. pf p ≤ buffer (ps $ p)) ∧
    tb = commonPushSubstrate cn ps pids (Max {n | p x q n. (p, x, q, n) ∈ sb}) pf ∧
    qs = commonPushQueues ps pf}

fun commonPullSubstrate where
  commonPullSubstrate sb qf = sb − (⋃ q. (qf q))

fun exact where
  exact [] s = (s = {})
| exact (x#xs) s = (∃(p, v, q, n) ∈ s. v = x ∧ exact xs (s − {(p, v, q, n)}))

fun commonPullItems where commonPullItems p s =
  { p(| buffer:= buffer p @ ys |) | ys. ys ∈ lists {v | p' v q n. (p', v, q, n) ∈ s} ∧ exact ys s }

fun commonPullQueues where
  commonPullQueues ps qf =
    { qs | qs. (dom qs = dom ps ∧ (∀ pid ∈ dom ps. qs $ pid ∈ commonPullItems (ps $ pid) (qf pid)))}

fun commonPull where
  commonPull sb ps cn =
  { (tb, qs) | tb qs qids qf.
    qids ⊆ dom ps ∧
    (∀ q ∈ qids. qf q ⊆ { (p, x, q', n) ∈ sb . q = q'}) ∧
    (∀ q ∈ UNIV − qids. qf q = {}) ∧
    (∀ q ∈ qids. card (qf q) + length (buffer (ps $ q)) ≤ capacity (ps $ q)) ∧
    tb = commonPullSubstrate sb qf ∧
    qs ∈ commonPullQueues ps qf}

definition CommonComm :: ((PortId * 'a::equal * PortId * nat) set, 'a) Communication where
  CommonComm = (|
    comPush= commonPush,
    comPull= commonPull
|)

end
```

Chapter 3

AADL Runtime Thread Dispatching Behavior

3.1 Thread Dispatch Logic

The specifications in this formalize AADL's rules for thread dispatching (as specified in Section 5.3.2 of the standard). Since our formalization does not yet consider time, some aspects of AADL's policies are not fully specified. In addition, the specifications below need to be doubled checked against the standard definitions (because I created these definitions more or less from memory following how HAMR is currently implemented).

This theory imports basic set and map definitions, Model (to access basic structures as well as declared thread and port properties related to dispatching, and ThreadState run time state to access the current state of infrastructure ports when considering sporadic thread dispatch.

theory *DispatchLogic*
  imports *SetsAndMaps Model ThreadState*
begin

datatype *EnabledStatus* =
    Periodic
  | *Sporadic PortId*

A periodic thread is enabled if the model-declared period of the thread (to be recorded in *CompDesc*) is appropriately related to current time (currently omitted). See the AADL Standard Section 5.3.2 (33).

This method is a placeholder. Its current implementation simply returns a value indicating that it is enabled

fun *periodicIsEnabled* :: *CompDescr* ⇒ *bool*
  where *periodicIsEnabled cd* = *True*

fun *periodicEnabledStatus* :: *CompDescr* ⇒ *EnabledStatus set*
  where *periodicEnabledStatus cd* = {*EnabledStatus.Periodic*}

A sporadic thread is enabled if two types of conditions are satisfied:

1. timing: the time interval since the last dispatch of the thread exceeds the model-declared "period" of the thread (the "period" attribute is slightly misnamed and actually reflects a minimum separation time). In the current state of the formalization, time is omitted and this condition is taken to be trivially true.

2. event arrival: at least one message/value has arrived on a port declared in the model as a dispatch trigger. By default (if no event triggers are explicitly declared in the model), all event and event data ports (see Section 5.4 (6))

Placeholder for to check if the minimum separation time for a sporadic thread is achieved.

fun *minSeparationAchieved* :: *Model* ⇒ *CompId* ⇒ *bool*
where *minSeparationAchieved m c* = *True*

selectMaximumUrgencyPorts returns the set of port ids from thread c in m that have the highest urgency among the given set of ports *candidateDispatchPort*. This function is intended to be called with the *candidateDispatchPorts* parameter instantiated to a set of input ports from c that have non-empty infrastructure queues (i.e., messages pending).

fun *selectMaximumUrgencyPorts* :: *Model* ⇒ *CompId* ⇒ *PortId set* ⇒ *PortId set*
where *selectMaximumUrgencyPorts m c candidateDispatchPorts* =
{*p* ∈ *portsOfCID m c* .∀ *p′* ∈ *candidateDispatchPorts* . *urgencyPID m p* ≥ *urgencyPID m p′*}

getDispatchablePorts returns the set of port ids from thread c in m are candidates for sporadic dispatching according to AADL's dispatching specification. Intuitively, a candidate most be declared (implicitly or explicitly in the AADL model) as a dispatch trigger, it must have data available in its infrastructure port state queue, and it must have the highest urgency among other candidate ports. Currently, if more than one port is declared with the same urgency, it is possible to have multiple dispatchable ports (whereas the standard specifies that there is at most one port available for dispatch). We still need to add the notion of AADL Queue_Processing_Protocol property, which would "break the tie" among multiple dispatchable ports. For now, we just return a set of ports and assume non-deterministic tie-breaking.

fun *getSporadicDispatchablePorts* :: *Model* ⇒ *CompId* ⇒ *CompDescr* ⇒ *′a PortState* ⇒ *PortId set*
where *getSporadicDispatchablePorts m c cd ps* =
(*let declaredDispatchTriggers* = *CompDescr.dispatchTriggers cd*
in let dataAvailableTriggers = {*p* ∈ *declaredDispatchTriggers* . *dataAvailablePID ps p*}
in selectMaximumUrgencyPorts m c dataAvailableTriggers)

The following two methods are not used directly in the semantics at the moment but are presented to enable traceability to the current version of the Slang reference semantics.

sporadicEnabledStatus returns a set of *EnabledStatus* values indicating which ports have triggered a sporadic dispatch. The fact that a set is currently returned instead of an indicator of a single port is a result of our currently not emphasizing the timestamp tie-breaking for sporadic dispatch candidates as reflected in AADL Queue Processing Protocol policy option. Our intention in HAMR is to eventually implement the FIFO option for the Queue Processing Protocol, which would return a single value that arrived earliest across all the non-empty ports which have the same urgency (see Section 8.3.2 (36)). For now, we assume that the client system semantics transitions non-deterministically pick from among the

returned set values. Note that AADL does allow a set of triggering ports to be made available to the Compute Entry Point user code (see Section 5.4.2 (39)).

```
fun sporadicEnabledStatus :: Model ⇒ CompId ⇒ CompDescr ⇒ 'a PortState ⇒ EnabledStatus set
  where sporadicEnabledStatus m c cd c-infi
    = (if (minSeparationAchieved m c)
       then (let dispatchablePorts = getSporadicDispatchablePorts m c cd c-infi
          in {es . ∃ p ∈ dispatchablePorts . es = Sporadic p})
       else {})
```

computeEnabledStatus returns a set of *EnabledStatus* values indicating if a thread is dispatchable. We first fetch the thread's dispatch protocol from the thread descriptor. Then, the return value is computed by calling helper methods for both Periodic and Sporadic dispatch protocol cases.

```
fun computeEnabledStatus :: Model ⇒ CompId ⇒ 'a PortState ⇒ EnabledStatus set
  where computeEnabledStatus m c ps =
   (   let compDescr = ((modelCompDescrs m) $ c)
    in let dp = CompDescr.dispatchProtocol compDescr
       in (case dp of
             DispatchProtocol.Periodic ⇒ periodicEnabledStatus compDescr
           | DispatchProtocol.Sporadic ⇒ sporadicEnabledStatus m c compDescr ps))
```

3.1.1 Determining Ports to Freeze

Once it is determined that a thread is dispatchable (enabled), we also determine which ports should be frozen in the dispatch action, because the information used to determine freeze ports is coupled to the information used to determine dispatch.

The functions/definitions in this section help compute the set of ports to freeze for a particular dispatch. This information is included in the *DispatchStatus* structure for both periodic and sporadic dispatches.

The rules for determining which ports to freeze are given primarily in Section 8.3.2, but also Section 5.4 (7), 5.4.2 (e.g., clauses 7,37), Section 8.3 (e.g., clause 2) of the standard. In general, the idea of AADL is that, upon dispatch, application code can only read (a) ports that are either non-dispatch triggers (see Section 8.3.2 (20) or (b) the port that was selected for sporadic dispatch. It is these ports that must be frozen on dispatch (we only support the Dispatch Time option for the AADL property Input Time, which indicates the point in time at which ports or frozen). The following is helpful intuition.

- Periodic threads – The notion of dispatch trigger is not relevant for Periodic threads (dispatch is only triggered by time-out, not event arrival). Therefore, all input ports are frozen (see e.g., Section 5.4.2 (33)).
- Sporadic threads – HAMR makes a few simplifying assumptions for Sporadic threads. First, it assumes that data ports can never be dispatch triggers (i.e., only event and event data ports can be triggers). The AADL standard actually allows data ports to be declared as dispatch triggers (i.e., triggered when fresh data arrives, see Section 5.4 (6), 5.4.2 (31)) but this is not currently implemented in HAMR. Then, following the standard for sporadic components, unless dispatch triggers are explictly declared as a thread property, all event and event data ports are assumed to be dispatch triggers in sporadic components. Practically speaking, this means that for Sporadic threads in HAMR, the typical situation is that all data ports will be frozen along with the event-like

port that triggered the dispatch (see the AADL Standard Section 5.3.2 (30). In the non-typical situation that only a subset of event-like ports are declared to be dispatch triggers, then all data ports, all non-trigger declared ports, and trigger-declared port that actually triggered the dispatch will be frozen. Intuitively, the application should not read non-frozen input ports. Right now, we have no way of enforcing that in the semantics, and we assume that this property is enforced at the code level (e.g., by static analysis).

```
fun getPeriodicPortsToFreeze :: Model ⇒ CompId ⇒ PortId set
  where getPeriodicPortsToFreeze m c = inPortsOfCID m c

fun getNonTriggeringEventLikePorts :: Model ⇒ CompId ⇒ PortId set
  where getNonTriggeringEventLikePorts m c = (inEventLikePortsOfCID m c) − (dispatchTriggersOfCID m c)

fun getSporadicAlwaysFreezePorts :: Model ⇒ CompId ⇒ PortId set
  where getSporadicAlwaysFreezePorts m c = (inDataPortsOfCID m c) ∪ (getNonTriggeringEventLikePorts m
c)

fun getSporadicPortsToFreeze :: Model ⇒ CompId ⇒ PortId ⇒ PortId set
  where getSporadicPortsToFreeze m c triggeringPort = (getSporadicAlwaysFreezePorts m c) ∪ {triggeringPort}
```

To Do: For a specific system, consider generating specialized lemmas and associated proofs that indicate that, e.g., a sporadic thread is enabled when there is an event on a specific port.

3.1.2 Computing Dispatch Status

```
fun computePeriodicDispatchStatus :: Model ⇒ CompId ⇒ CompDescr ⇒ (DispatchStatus set)
  where computePeriodicDispatchStatus m c cd =
    (if periodicIsEnabled cd
     then (let portsToFreeze = getPeriodicPortsToFreeze m c
           in {DispatchStatus.Periodic portsToFreeze})
     else {DispatchStatus.NotEnabled})

fun computeSporadicDispatchStatus ::
  Model ⇒ CompId ⇒ CompDescr ⇒ 'a PortState ⇒ (DispatchStatus set)
  where computeSporadicDispatchStatus m c cd c-infi =
    (let dispatchablePorts = getSporadicDispatchablePorts m c cd c-infi
      in {ds . ∃ p ∈ dispatchablePorts .
                ds = (DispatchStatus.Sporadic (p, getSporadicPortsToFreeze m c p))})


fun computeDispatchStatus :: Model ⇒ CompId ⇒ 'a PortState ⇒ (DispatchStatus set)
  where computeDispatchStatus m c c-infi =
   (   let compDescr = ((modelCompDescrs m) $ c)
    in let dp = CompDescr.dispatchProtocol compDescr
       in (case dp of
             DispatchProtocol.Periodic ⇒ computePeriodicDispatchStatus m c compDescr
           | DispatchProtocol.Sporadic ⇒ computeSporadicDispatchStatus m c compDescr c-infi))
```

end

Chapter 4

Thread Application Behavior

4.1 Relational Behavior of Thread Application Logic

theory *App*
imports *Main VarState PortState ThreadState Model*
begin

4.1.1 Application Logic Relations

The application code for an AADL Thread component is organized into entry points. This semantics currently supports the Initialization and Compute entry points. In the HAMR/GUMBO framework, GUMBO contracts are attached to AADL Thread component interfaces, and then thread entry point code is shown to conform to the contracts via SMT-based verification in Logika or via testing.

Roughly speaking, each entry point contract specifies a a relation between component input ports and output ports and also characterizes how the thread's local variable state is transformed. GUMBO contracts are automatically generated (planned) by HAMR into Isabelle as predicates on thread/port state elements. These predicates are represented as a shallow embedding as reflected in the *InitContract* and *ComputeContract* types shown below. These predicates are then lifted to set-based definitions of component behaviors reflected in the *InitRelation* and *ComputeRelation* types shown below.

Intuitively, the Initialize entry point takes as inputs the initial var state for the thread, where the value of each variable is set to an unspecified system default value, and it gives each thread value an initial value. In addition, it gives each output data port a value, and may optionally give each event-like port a value.

Discuss / To Do: - it's likely that as a simplifying assumption, we should require each output data port to be given a value. This should be stated in the well-formed condition.

The Compute entry point takes as inputs the current var state, application input port state, and dispatch status. The dispatch status information is produced by the AADL RunTime dispatch logic and indicates what triggered the dispatch of the thread. This is most relevant for Sporadic threads where the dispatch status indicates the port that triggered the dispatch due to event arrival on that port. Conceptually, this allows the application logic to select behavior specific to the triggering event, e.g., in code corresponding to an event handler. The Compute entry point produces an updated var state where individual variables

are optionally updated and where output ports are optionally given values.

Below are the types for the shallow embedding predicate-based representations of GUMBO contracts.

type-synonym *'a InitContract* = (*'a PortState* ⇒ *'a VarState* ⇒ *bool*)
type-synonym *'a ComputeContract* = (*'a PortState* ⇒ *'a VarState* ⇒
DispatchStatus ⇒
'a PortState ⇒ *'a VarState* ⇒ *bool*)

Below are the types for the set-based representations of component behaviors (which will be derived from the predicate representation of the contracts).

type-synonym *'a InitBehavior* = (*'a PortState* ⇒ *'a VarState* ⇒ *bool*)
type-synonym *'a ComputeBehavior* = (*'a PortState* ⇒ *'a VarState* ⇒
DispatchStatus ⇒
'a PortState ⇒ *'a VarState* ⇒ *bool*)

The following type represents (abstractly) the behavior of the thread component's application code as derived from its contracts.

record *'a AppBehavior* =
appInit :: *'a InitBehavior*
appCompute :: *'a ComputeBehavior*

The following definitions "lift" the predicate/contract-based representation into a set-based representation.

The strategy for handling contract well-formedness issues is important and was the subject of much deliberation. From a practical/implementation point of view, HAMR will check that each GUMBO contract is well-formed in the sense that:

- it only reference features (ports, variables) from component to which it belongs,
- it doesn't confuse input and output ports,
- it doesn't confuse types associated with ports and variables.

With a shallow embedding representation of contracts, there is no way to directly check (confirm) those properties in the Isabelle representation. For example, to check type correctness, we would need a formalization of predicate expression ASTs and an associated type checker.

Ultimately, we desire that, given a well-formed input *ThreadState*, the relational app behavior of a component will yield well-formed output *ThreadState*s. This will enable us support the fundamental run-time property that each thread execution step in the semantics preserves well-formed thread-states. Note that HAMR should also guarantee this, and we eventually want to demonstrate this by showing HAMR state logs conform with Isabelle system execution steps.

In any case, the following definitions are currently designed so that, in lifting from the predicate representations of the contracts, the predicates are only applied to well-formed input and output thread states.

Therefore, we would expect that the presence of mal-formed HAMR contracts (manifested in mal-formed predicates) would give rise to empty relations.

An alternative approach that needs to be investigated is that we only assume input states are well-formed, and we prove that, for a given contract, output states are well-formed. This will require that we add frame-conditions to the contracts (intuitively, capturing implicit properties that HAMR ensures during entry point execution).

```
definition initInfrastructureContract :: Model ⇒ CompId ⇒ 'a InitContract
  where initInfrastructureContract m cid ao tv ≡
            wf-ThreadState-appo m cid ao
        ∧   wf-ThreadState-tvar m cid tv



definition computeInfrastructureContract :: Model ⇒ CompId ⇒ 'a ComputeContract
  where computeInfrastructureContract m cid ai tvi ds ao tvo ≡
          wf-ThreadState-appi m cid ai
        ∧ wf-ThreadState-tvar m cid tvi
        ∧ wf-ThreadState-disp m cid ds
        ∧ wf-ThreadState-appo m cid ao
        ∧ wf-ThreadState-tvar m cid tvo


fun mk-InitBehaviorFromContract :: Model ⇒ CompId ⇒
   'a InitContract ⇒ 'a InitBehavior where
  mk-InitBehaviorFromContract m cid initContract ao tvo =
            (initContract ao tvo
        ∧ initInfrastructureContract m cid ao tvo)

fun mk-ComputeBehaviorFromContract :: Model ⇒ CompId ⇒
    'a ComputeContract
 ⇒ 'a ComputeBehavior where
  mk-ComputeBehaviorFromContract m cid computeContract ai tvi d ao tvo  =
          (computeContract ai tvi d ao tvo
       ∧  computeInfrastructureContract m cid ai tvi d ao tvo)

fun mk-AppBehaviorFromContracts ::
  Model ⇒ CompId ⇒ 'a InitContract ⇒ 'a ComputeContract ⇒ 'a AppBehavior where
  mk-AppBehaviorFromContracts m cid initUserContract computeUserContract = (|
    appInit = mk-InitBehaviorFromContract m cid initUserContract,
    appCompute = mk-ComputeBehaviorFromContract m cid computeUserContract |)
```

An *App a* is well-formed wrt a *Model m* and *CompId c* iff the thread state elements associated with the relations (transfer functions) of *a* are well-formed wrt *m* and *c*.

```
definition wf-ThreadState-dataPorts :: Model ⇒ CompId ⇒ 'a PortState ⇒ bool
  where wf-ThreadState-dataPorts m c ps ≡
        ∀ p ∈ dom ps . isDataPID m p ⟶ isOneElement (ps $ p)

definition wf-InitBehavior:: Model ⇒ CompId ⇒ 'a InitBehavior ⇒ bool
  where wf-InitBehavior m c initBehavior ≡
    (∀ ao tvo . initBehavior ao tvo ⟶
```

```
    (  wf-ThreadState-appo m c ao
     ∧ wf-ThreadState-tvar m c tvo
     ∧ wf-ThreadState-dataPorts m c ao))

definition wf-InitBehavior-Inhabited:: Model ⇒ CompId ⇒ 'a InitBehavior ⇒ bool
  where wf-InitBehavior-Inhabited m c initBehavior ≡
    (∃ ao tvo . initBehavior ao tvo)

definition wf-ComputeBehavior:: Model ⇒ CompId ⇒ 'a ComputeBehavior ⇒ bool
  where wf-ComputeBehavior m c computeBehavior ≡
    (∀ ai tvi ds ao tvo . computeBehavior ai tvi ds ao tvo ⟶
      (  wf-ThreadState-appi m c ai
       ∧ wf-ThreadState-dataPorts m c ai
       ∧ wf-ThreadState-tvar m c tvi
       ∧ wf-ThreadState-disp m c ds
       ∧ wf-ThreadState-appo m c ao
       ∧ wf-ThreadState-dataPorts m c ao
       ∧ wf-ThreadState-tvar m c tvo
       ))

definition wf-ComputeBehavior-Inhabited:: Model ⇒ CompId ⇒ 'a ComputeBehavior ⇒ bool
  where wf-ComputeBehavior-Inhabited m c computeBehavior ≡
    (∃ ai tvi ds ao tvo . computeBehavior ai tvi ds ao tvo)

definition wf-AppBehavior:: Model ⇒ CompId ⇒ 'a AppBehavior ⇒ bool
  where wf-AppBehavior m c a ≡
    c ∈ dom (modelCompDescrs m)
  ∧ wf-InitBehavior m c (appInit a)
  ∧ wf-InitBehavior-Inhabited m c (appInit a)
  ∧ wf-ComputeBehavior m c (appCompute a)
  ∧ wf-ComputeBehavior-Inhabited m c (appCompute a)
```

Each thread component is associated with its application logic via an *AppBehaviors* structure – a map from component identifiers to application code behaviors.

```
type-synonym 'a AppBehaviors = (CompId, 'a AppBehavior) map
```

A *AppBehaviors* structure is well-formed, if each *AppBehavior* is well-formed wrt the associated component.

```
definition wf-AppBehaviors where wf-AppBehaviors m cb ≡ ∀ c. wf-AppBehavior m c (cb $ c)

record 'a BehaviorModel =
  bmmodel :: Model
  bmbehaviors :: 'a AppBehaviors

definition wf-BehaviorModel where
  wf-BehaviorModel bm ≡
      wf-Model (bmmodel bm)
    ∧ wf-AppBehaviors (bmmodel bm) (bmbehaviors bm)
```

The following helper functions are defined to access elements of a *BehaviorModel*.

fun *bmodelCompDescrs* **where** *bmodelCompDescrs bm* = *modelCompDescrs* (*bmmodel bm*)
fun *bmodelPortDescrs* **where** *bmodelPortDescrs bm* = *modelPortDescrs* (*bmmodel bm*)
fun *bmodelPortKind* **where** *bmodelPortKind bm p* = *kind* ((*bmodelPortDescrs bm*) $ *p*)
fun *bmodelConns* **where** *bmodelConns bm* = *modelConns* (*bmmodel bm*)
fun *bmodelCIDs* **where** *bmodelCIDs bm* = *dom* (*bmodelCompDescrs bm*)
fun *bmodelPIDs* **where** *bmodelPIDs bm* = *dom* (*bmodelPortDescrs bm*)
fun *bmodelInit* **where** *bmodelInit bm c* = *appInit* (*bmbehaviors bm* $ *c*)
fun *bmodelCompute* **where** *bmodelCompute bm c* = *appCompute* (*bmbehaviors bm* $ *c*)

end

Chapter 5

System Behavior

theory *Behavior*
imports *SetsAndMaps Model App ThreadState SystemState DispatchLogic RTS*
begin

5.1 Thread Execution

The rules in this section reflect how a thread's state is transformed by executing the application logic for thread entry points. The state transformation for each entry point is determined by the transfer function (relation) derived from the entry point contract as defined in App.thy. The application logic transfer relation is defined on the portions of the thread state that are visible to the application code: the application input port state *appi*, the application output port state *appo* and the thread variables *tvar*. The rules in this section "lift" the transfer relation from the application-code-visible portions of the state to the entire thread state. In addition, for Compute entry point execution, the rules include invocations of AADL run-time services to manage the enqueuing and dequeuing of port queues.

5.1.1 Initialize Entry Point

The Initialize entry point does not read any portion of the state; it only gives initial values for output ports and thread variables. Therefore, the app logic "transfer relation" *appInit* degenerates to a predicate on the application output port state and thread variables. The rule below "lifts" that predicate to an "update" the entire thread state. Only the elements *vs* and *ps* are updated. The other elements of the thread state are unchanged.

inductive *stepInit* **for** $a :: 'a$ *AppBehavior* **where**
initialize: *appInit a ao tvo* $\Longrightarrow$ *stepInit a t* (t(| *appo*:= *ao*, *tvar*:= *tvo* |))

Below a rule inversion property is proved that states that if the thread can do a *stepInit* step, it must be the case that the execution of the thread Initialize application behavior *appInit* could produce the values of the thread variable state and application output port state.

lemma *stepInit-ruleinv*:
assumes *stepInit a t* t'

```
  shows appInit a (appo t') (tvar t')
proof −
  obtain ao tvo where h1: t' = t(|appo := ao, tvar := tvo|)
                  and h2: appInit a ao tvo
    using assms by (metis stepInit.cases)
  have h3: tvo = tvar t' using h1
    apply (drule-tac f = tvar in arg-cong)
    by simp
  have h4: ao = appo t' using h1
    apply (drule-tac f = appo in arg-cong)
    by simp
  thus ?thesis using h2 h3 h4 by blast
qed



inductive stepThread for cds :: 'a PortState ⇒ DispatchStatus set
                and pki :: PortId ⇒ PortKind
                and app :: 'a AppBehavior
                and cat :: ScheduleState * ScheduleState
                where


  dispatch: ⟦ cat = (Waiting, Ready);
              dsp ∈ cds (infi t);
              dsp ≠ DispatchStatus.NotEnabled;
              receiveInput pki (dispatchInputPorts dsp) (infi t) (appi t) infi' appi' ⟧
    ⟹ stepThread cds pki app cat t (t(| infi:= infi', appi:= appi', disp:= dsp |))


| compute: ⟦ cat = (Ready, Running); appCompute app (appi t) (tvar t) (disp t) ao tvo ⟧
    ⟹ stepThread cds pki app cat t (t(| appi:= clearAll (dom (appi t)) (appi t),
                                        appo:= ao,
                                        tvar:= tvo |))

| complete: ⟦ cat = (Running, Waiting); sendOutput (appo t) (info t) appo' info' ⟧
    ⟹ stepThread cds pki app cat t (t(| appo:= appo', info:= info', disp:= DispatchStatus.NotEnabled |))
```

5.2 System Execution

```
definition initSys where initSys bm sc s ≡
  (wf-SystemState (bmmodel bm) s) ∧
  (systemExec s = Initialize (scheduleInit sc)) ∧
  (∀ c ts. inModelCID (bmmodel bm) c ∧ systemThread s c = Some ts ⟶ initial-ThreadState (bmmodel bm) c
ts) ∧
  (∀ c x. systemState s c = Some x ⟶ x = Waiting) ∧
  (systemComms s = {})
```

inductive *initStepSys* **for** *bm* :: *'a BehaviorModel*
and *cm* :: *('u, 'a) Communication*
and *sc* :: *SystemSchedule*

where

initialize: ⟦ *isInitializing s*; *systemExec s = Initialize (c#cs)*;
stepInit (bmbehaviors bm $ c) (systemThread s $ c) t ⟧ ⟹
initStepSys bm cm sc s (s(| systemThread:= (systemThread s)(c↦t), systemExec:= Initialize cs |))

| *switch*: ⟦ *isInitializing s*; *systemExec s = Initialize []*; *c ∈ scheduleFirst sc* ⟧
⟹ *initStepSys bm cm sc s (s(| systemExec:= Compute c |))*

inductive *computeStepSys* **for** *bm* :: *'a BehaviorModel*
and *cm* :: *('u, 'a) Communication*
and *sc* :: *SystemSchedule*

where

push: ⟦ *isComputing s*; *systemThread s c = Some t*;
(sb, it) ∈ comPush cm (systemComms s) (info t) (modelConns (bmmodel bm)) ⟧
⟹ *computeStepSys bm cm sc s (s(|*
systemThread:= (systemThread s)(c↦(t(| info:= it |))),
systemComms:= sb |))

| *pull*: ⟦ *isComputing s*; *systemThread s c = Some t*;
(sb, it) ∈ comPull cm (systemComms s) (infi t) (modelConns (bmmodel bm)) ⟧
⟹ *computeStepSys bm cm sc s (s(|*
systemThread:= (systemThread s)(c↦(t(| infi:= it |))),
systemComms:= sb |))

| *execute*: ⟦ *isComputing s*;
systemExec s = Compute c;
c' ∈ scheduleComp sc $ c;
stepThread (computeDispatchStatus (bmmodel bm) c)
(bmodelPortKind bm)
(bmbehaviors bm $ c)
(systemState s $ c, a)
(systemThread s $ c) t ⟧
⟹ *computeStepSys bm cm sc s (s(|*
systemThread:= (systemThread s)(c↦t),
systemState:= (systemState s)(c↦a),
systemExec:= Compute c' |))

lemma *compute-not-initialize*:
assumes *comp*: *computeStepSys bm cm sc s s'*
shows *¬isInitializing s*
using *assms computeStepSys.cases init-not-compute* **by** *blast*

lemma *initialize-not-compute*:
assumes *init*: *initStepSys bm cm sc s s'*
shows ¬*isComputing s*
using *assms initStepSys.simps init-not-compute* **by** *blast*

definition *stepSys* **where**
stepSys bm cm sc s s' ≡ *initStepSys bm cm sc s s'* ∨ *computeStepSys bm cm sc s s'*

lemma *stepSys-init*: *initStepSys bm cm sc s s'* ⟹ *stepSys bm cm sc s s'* **by** (*simp add*: *stepSys-def*)

lemma *stepSys-init-rtranclp*: (*initStepSys bm cm sc*)** *s s'* ⟹ (*stepSys bm cm sc*)** *s s'*
by (*metis mono-rtranclp stepSys-init*)

lemma *stepSys-compute*: *computeStepSys bm cm sc s s'* ⟹ *stepSys bm cm sc s s'* **by** (*simp add*: *stepSys-def*)

lemma *stepSys-compute-rtranclp*: (*computeStepSys bm cm sc*)** *s s'* ⟹ (*stepSys bm cm sc*)** *s s'*
by (*metis mono-rtranclp stepSys-compute*)

lemma *stepSys-initializing*:
assumes *init*: *isInitializing s*
shows *stepSys bm cm sc s s'* = *initStepSys bm cm sc s s'*
unfolding *stepSys-def*
using *compute-not-initialize init* **by** *blast*

lemma *stepSys-initializing-back*:
assumes *init*: *isInitializing s'*
and *step*: *stepSys bm cm sc s s'*
shows *isInitializing s*
proof −
have *computeStepSys bm cm sc s s'* ⟹ ¬*isInitializing s'*
proof (*induction rule*: *computeStepSys.induct*)
case (*push c t sb it*)
then show *?case* **apply** *simp* **by** *fastforce*
next
case (*pull c t sb it*)
then show *?case* **apply** *simp* **by** *fastforce*
next
case (*execute c c' a t*)
then show *?case* **by** *simp*
qed
hence ¬*computeStepSys bm cm sc s s'* **using** *init* **by** *blast*
hence *h2*: *initStepSys bm cm sc s s'* **using** *step* **unfolding** *stepSys-def* **by** *blast*
thus *?thesis* **using** *initStepSys.simps* **by** *blast*
qed

lemma *stepSysInit-ruleinv*:
assumes *init*: *isInitializing s*
and *exec*: *systemExec s* = *Initialize* (*x* # *xs*)
and *step*: *stepSys bm cm sc s s'*
shows *stepInit* (*bmbehaviors bm* $ *x*) (*systemThread s* $ *x*) (*systemThread s'* $ *x*)

proof −
— First show that *initStepSys* rule must have been applied, because the *Initialize* schedule is not empty.
from *assms* **have** *h1*: *initStepSys bm cm sc s s′* **using** *stepSys-initializing* **by** *blast*
— Then the result follows by rule cases.
from *init exec h1* **show** *?thesis* **using** *initStepSys.cases* **by** *fastforce*
qed

lemma *stepSysInit-redsch-ruleinv*:
assumes *init*: *isInitializing s*
and *exec*: *systemExec s* = *Initialize* (*x* # *xs*)
and *step*: *stepSys bm cm sc s s′*
shows *systemExec s′* = *Initialize xs*
proof −
— The *initStepSys* rule must have been applied, because the *Initialize* schedule is not empty.
from *assms* **have** *h1*: *initStepSys bm cm sc s s′* **using** *stepSys-initializing* **by** *blast*
— Then the result follows by rule cases.
from *init exec h1* **show** *?thesis* **using** *initStepSys.cases* **by** *fastforce*
qed

lemma *stepSysInit-initInv-ruleinv*:
assumes *init*: *isInitializing s*
and *exec*: *systemExec s* = *Initialize* (*x* # *xs*)
and *step*: *stepSys bm cm sc s s′*
shows *isInitializing s′*
proof −
— The *initStepSys* rule must have been applied, because the *Initialize* schedule is not empty.
from *assms* **have** *h1*: *initStepSys bm cm sc s s′* **using** *stepSys-initializing* **by** *blast*
— Then the result follows by rule cases.
from *init exec h1* **show** *?thesis* **using** *initStepSys.cases* **by** *fastforce*
qed

lemma *stepSysInit-sc-rev-ruleinv*:
assumes *step*: *stepSys bm cm sc s s′*
and *exec*: *systemExec s′* = *Initialize xs*
shows ∃ *x*. *systemExec s* = *Initialize* (*x* # *xs*)
proof −
— Since s' is initializing, and we took a step, then s is initializing.
from *assms* **have** *h1*: *isInitializing s* **using** *stepSys-initializing-back init-init* **by** *blast*
— We must have taken an init step from s to s'.
from *assms h1* **have** *h2*: *stepSys bm cm sc s s′* = *initStepSys bm cm sc s s′*
using *stepSys-initializing* **by** *blast*
from *h2* **have** *h3*: *initStepSys bm cm sc s s′* **using** *step* **unfolding** *stepSys-def* **by** *simp*
— ..and then the result follows by cases.
from *h1 h3 exec* **show** *?thesis* **using** *initStepSys.cases* **by** *fastforce*
qed

definition *stepsSys* **where** *stepsSys bm cm sc* = (*stepSys bm cm sc*)**
definition *initStepsSys* **where** *initStepsSys bm cm sc* = (*initStepSys bm cm sc*)**

definition *computeStepsSys* **where** *computeStepsSys bm cm sc* = (*computeStepSys bm cm sc*)**

definition *reachSys* **where** *reachSys bm cm sc y* ≡
  ∃ *x*. *initSys bm sc x* ∧ *stepsSys bm cm sc x y*

end

Chapter 6

Verification Framework

6.1 Deductive Schemas

Definitions in this section set up schema for reasoning about system execution properties. One role of the schema is to state how component-level properties to support system-level properties.

theory *Properties*
imports *Behavior*
begin

6.1.1 Initialization Phase

The following definition introduces a notion of a *thread initialization property* for thread application logic (which abstracts thread entry point code). The property P holds for a thread's application logic if it holds for all output variable states vs' and output port states ps' that satisfy the thread's Initialize entry point behavior (*appInit*).

definition *appInitProp* :: *'a AppBehavior* ⇒ (*'a PortState* ∗ *'a VarState* ⇒ *bool*) ⇒ *bool*
where *appInitProp a P* ≡ ∀ *ao two*. *appInit a ao two* ⟶ *P* (*ao*, *two*)

We introduce the notion a *system initialization property*. Currently, the only system features that can be "observed" by a system property are the variable states and application port states of a thread. A system initialization property makes observations about the results of "executing" the Initialization entry points of each thread (which only constrain variable states and output application port states. Therefore, the system initialization property is intuitively a family of properties indexed by component identifier c where each member of the family observes the variables states and output application port states for a given component/thread c.

The following definition states that a system initialization property P holds for the app model am portion of a system when, for all threads (identifiers) c, if a thread can undergo an initialize step from thread state t to thread state t', then the c-relevant portion of the property holds for the variable state *tvar* and output application port state *appo* elements of thread state t'.

definition *sysInitProp* :: *'a BehaviorModel* ⇒ (*CompId* ⇒ (*'a PortState* ∗ *'a VarState* ⇒ *bool*)) => *bool*
where *sysInitProp bm P* ≡

$\forall\, c \in modelCIDs\ (bmmodel\ bm).\ \forall\, (t::'a\ ThreadState)\ t'.\ stepInit\ (bmbehaviors\ bm\ \$\ c)\ t\ t' \longrightarrow P\ c\ (appo\ t',\ tvar\ t')$

Now, we set up a verification condition *sysInitVC* for a system initialization property P. We intend to show that, to verify a system initialization property P (i.e., to show that P holds for an app model *am*, it is sufficient to show that for every thread component c in the model, the c-relevant portion of P is an thread initialization property (*appInitProp*) for c (i.e., for c's application logic).

definition *sysInitVC* :: $'a\ BehaviorModel \Rightarrow (CompId \Rightarrow ('a\ PortState * 'a\ VarState\ \Rightarrow bool)) => prop$
where *sysInitVC bm P* ≡
$(\bigwedge c.\ c \in modelCIDs\ (bmmodel\ bm) \Longrightarrow appInitProp\ (bmbehaviors\ bm\ \$\ c)\ (P\ c))$

The following lemma establishes that *sysInitVC* is a sound verification condition for system initialization property P: for a well-formed app model *am*, if *sysInitVC* holds, then *sysInitProp* holds.

lemma *initSysFromApps*:
assumes *wf-bm*: *wf-BehaviorModel* (*bm*::$'a$ *BehaviorModel*)
and *vc*: $\bigwedge c.\ c \in modelCIDs\ (bmmodel\ bm) \Longrightarrow appInitProp\ (bmbehaviors\ bm\ \$\ c)\ (P\ c)$
shows *sysInitProp bm P*
proof (*simp only*: *sysInitProp-def*; *clarify*)
fix *c*
fix *t t'*::$'a$ *ThreadState*
assume *a1*: $c \in modelCIDs\ (bmmodel\ bm)$
and *a2*: $stepInit\ (bmbehaviors\ bm\ \$\ c)\ t\ t'$
thus $P\ c\ (appo\ t',\ tvar\ t')$
proof −
have $appInit\ (bmbehaviors\ bm\ \$\ c)\ (appo\ t')\ (tvar\ t')$
using *a2 stepInit-ruleinv*[*of* $(bmbehaviors\ bm\ \$\ c)\ t\ t'$] **by** *blast*
thus *?thesis*
using *a1 appInitProp-def vc* **by** *blast*
qed
qed

The following definition will interpret a system initialization property P in the context of a specific system state *s*.

definition *sysStateProp* :: $'a\ BehaviorModel \Rightarrow$
$(CompId \Rightarrow ('a\ PortState * 'a\ VarState \Rightarrow bool)) \Rightarrow$
$('u,\ 'a)\ SystemState \Rightarrow bool$
where *sysStateProp bm P s* ≡
$\forall\, c \in modelCIDs\ (bmmodel\ bm).\ P\ c\ (appo\ (systemThread\ s\ \$\ c),\ tvar\ (systemThread\ s\ \$\ c))$

The following definition forms the set of all possible Initialization Entrypoint outputs of thread component c, where outputs are pairs of var states (*tvar*) v and application output port states (*appo*) *ao*.

definition *systemAppInit* :: $'a\ BehaviorModel \Rightarrow CompId \Rightarrow ('a\ PortState \times 'a\ VarState\ )\ set$
where $systemAppInit\ bm\ c = \{\ (ao,\ tvo)\ |\ ao\ tvo.\ appInit\ (bmbehaviors\ bm\ \$\ c)\ ao\ tvo\ \}$

The following definition projects a state to a tuple consisting of thread app output port states (*appo*) and thread variable states (*tvar*) – the two thread state elements affected by thread initialisation.

definition *appovar* :: $('u,\ 'a)\ SystemState \Rightarrow CompId \Rightarrow ('a\ PortState \times 'a\ VarState)\ option$
where *appovar x c* =

```
  (if c ∈ dom (systemThread x)
    then Some (appo (systemThread x $ c), tvar (systemThread x $ c))
    else None)

lemma appovar-te:
  assumes c ∈ dom(systemThread x)
  shows appovar (x(|systemThread :=
    (systemThread x)(c ↦ (systemThread x $ c)(|appo := ao, tvar := tvo|)), systemExec := e|)) c =
    Some (ao, tvo)
proof −
  have h1: tvar((systemThread x)(c ↦ (systemThread x $ c)(|appo := ao, tvar := tvo|)) $ c) = tvo
    by simp
  have h2: appo((systemThread x)(c ↦ (systemThread x $ c)(|appo := ao, tvar := tvo|)) $ c) = ao
    by simp
  show ?thesis unfolding appovar-def by simp
qed

lemma sysInit-bw:
  shows ⟦stepSys bm com sc x y; systemExec y = Initialize cs ⟧ ⟹ systemExec x ≠ Initialize []
  using stepSysInit-sc-rev-ruleinv by fastforce

lemma sysInit-seq-bw-neq:
  ⟦(stepSys bm com sc)** x y; x ≠ y; systemExec y = Initialize cs⟧ ⟹ systemExec x ≠ Initialize []
proof (induction arbitrary: cs rule: rtranclp.induct)
  case (rtrancl-refl a)
  then show ?case by blast
next
  case (rtrancl-into-rtrancl a b c)
  have h1: (stepSys bm com sc)** a b using rtrancl-into-rtrancl.hyps(1) by blast
  have h2: stepSys bm com sc b c using rtrancl-into-rtrancl.hyps(2) by blast
  have h3: ⋀cs. a ≠ b ⟹ systemExec b = Initialize cs ⟹ systemExec a ≠ Initialize []
    using rtrancl-into-rtrancl.IH by blast
  have h4: a ≠ c by (simp add: rtrancl-into-rtrancl.prems(1))
  have h5: systemExec c = Initialize cs using rtrancl-into-rtrancl.prems(2) by blast
  show ?case
  proof (cases a = b)
    case True
    then show ?thesis
      using rtrancl-into-rtrancl.hyps(2) rtrancl-into-rtrancl.prems(2) sysInit-bw by blast
  next
    case False
    then obtain cs' where systemExec b = Initialize cs'
      using h2 h5 stepSysInit-sc-rev-ruleinv by blast
    then show ?thesis using False h3 by blast
  qed
qed

lemma sysInit-seq-bw-supseq:
  ⟦(stepSys bm com sc)** x y; systemExec y = Initialize cs ⟧ ⟹ (∃as. systemExec x = Initialize (as @ cs))
proof (induction arbitrary: cs rule: rtranclp.induct)
```

```
  case (rtrancl-refl a)
  then show ?case by simp
next
  case (rtrancl-into-rtrancl a b c)
  then show ?case
    by (metis append.assoc append-Cons append-Nil stepSysInit-sc-rev-ruleinv)
qed

lemma sysInit-none:
  ⟦stepSys bm com sc x y; systemExec y = Initialize cs; systemExec x = Initialize cs ⟧ ⟹ x = y
  using stepSysInit-sc-rev-ruleinv by fastforce

lemma sysInit-seq-none:
  ⟦(stepSys bm com sc)** x y; systemExec y = Initialize cs; systemExec x = Initialize cs ⟧ ⟹ x = y
proof (induction rule: rtranclp.induct)
  case (rtrancl-refl a)
  then show ?case by simp
next
  case (rtrancl-into-rtrancl a b c)
  obtain u where systemExec b = Initialize (u#cs)
    using rtrancl-into-rtrancl.hyps(2) rtrancl-into-rtrancl.prems(1) stepSysInit-sc-rev-ruleinv by blast
  then show ?case
    using rtrancl-into-rtrancl.hyps(1) rtrancl-into-rtrancl.prems(2) sysInit-seq-bw-supseq by fastforce
qed

lemma sysInit-step-seq-set:
  assumes step: stepSys bm com sc x y
      and exec: systemExec x = Initialize (c # cs)
    shows appovar y ∈ { (appovar x)(c↦v) |v. v ∈ systemAppInit bm c }
proof −
  have stepSys bm com sc x y = initStepSys bm com sc x y
    using stepSys-initializing exec init-init by blast
  hence initStepSys bm com sc x y using step by force
  thus ?thesis using exec
  proof (induction rule: initStepSys.induct)
    case (initialize s c cs t)
    obtain ps vs where h2: appovar (s(| systemThread := (systemThread s)(c ↦ t),
                                        systemExec := Initialize cs|)) c = Some (ps, vs)
      using appovar-def domI fun-upd-same
      by (metis SystemState.SystemState.simps(1) SystemState.SystemState.simps(9)
          SystemState.SystemState.surjective SystemState.SystemState.update-convs(1))
    hence h3: (ps, vs) ∈ systemAppInit bm c unfolding systemAppInit-def appovar-def apply clarify
      using initialize.hyps(3) stepInit-ruleinv by fastforce
    have h4: appovar (s(|systemThread := (systemThread s)(c ↦ t),
                         systemExec := Initialize cs|)) = (appovar s)(c ↦ (ps, vs))
    proof
      fix z
      show appovar (s(|systemThread := (systemThread s)(c ↦ t), systemExec := Initialize cs|)) z =
           ((appovar s)(c ↦ (ps, vs))) z
      proof (cases z = c)
```

```
      case True
      then show ?thesis
        by (simp add: h2)
    next
      case False
      then show ?thesis by (simp add: appovar-def)
    qed
  qed
  then show ?case using exec h3 using initialize.hyps(2) initialize.prems by auto
 next
  case (switch c)
  then show ?case by fastforce
 qed
qed

lemma sysInit-step-seq:
 assumes step: stepSys bm com sc x y
    and exec: systemExec x = Initialize (c # cs)
  shows appovar y ∈ map-Upd-seq (systemAppInit bm) {appovar x} [c]
proof −
 have stepSys bm com sc x y = initStepSys bm com sc x y
  using step stepSys-initializing exec init-init by blast
 hence h0: initStepSys bm com sc x y using step by force
 thus ?thesis using exec
 proof (induction rule: initStepSys.induct)
  case (initialize s a as t)
   have h1: a = c using initialize.hyps(2) initialize.prems(1) by fastforce
   have h2: as = cs using initialize.hyps(2) initialize.prems(1) by fastforce
    have h3: map-Upd-seq (systemAppInit bm) {appovar s} [c] = { (appovar s)(c↦v) |v. v ∈ systemAppInit
bm c }
     by simp
   show ?case
    using step exec h0 sysInit-step-seq-set[of bm com sc s y c cs]
      initStepSys.initialize[of s c cs bm t] initialize.hyps(1) initialize.hyps(3)
    apply (simp add: h1 h2 h3)
    using stepSys-initializing sysInit-step-seq-set
    by (smt (verit, best) ‹⋀sc cm. ⟦isInitializing s; systemExec s = Initialize (c # cs); stepInit (bmbehaviors
bm $ c) (systemThread s $ c) t⟧ ⟹ initStepSys bm cm sc s (s(|systemThread := (systemThread s) (c ↦
t), systemExec := Initialize cs|))› h1 h2 initialize.hyps(1) initialize.hyps(2) initialize.hyps(3) mem-Collect-eq
old.prod.exhaust)
  next
   case (switch c)
    show ?case using exec switch.hyps(2) by (simp add: switch.prems)
  qed
qed
```

Lemma sysInit_seq is used to prove lemma sysInit_init_seq by induction.

```
lemma sysInit-seq:
 ⟦ stepsSys bm com sc x y;
   systemExec x = Initialize xs; systemExec y = Initialize [] ⟧ ⟹
```

```
    appovar y ∈ map-Upd-seq (systemAppInit bm) {appovar x} xs
proof (induction xs arbitrary: x)
  case Nil
  then show ?case unfolding appovar-def stepsSys-def using sysInit-seq-none by fastforce
next
  case (Cons a xs)
  have h1: isInitializing x by (simp add: Cons.prems(2))
  obtain z where z1: stepSys bm com sc x z
            and z2: stepsSys bm com sc z y
    using Cons.prems unfolding stepsSys-def
    by (metis Exec.inject(1) converse-rtranclpE list.distinct(1))
    have z3: systemExec z = Initialize (xs) using z1 h1 Cons.prems(2) stepSysInit-redsch-ruleinv by blast
    have z4: appovar z ∈ map-Upd-seq (systemAppInit bm) {appovar x} [a]
      using sysInit-step-seq Cons.prems(3) z1 Cons.prems(2) by blast
  show ?case using Cons.prems Cons.IH[of z] z1 z2 z3 z4
    by (metis (no-types, opaque-lifting) append-Cons append-Nil map-Upd-seq-comp-in)
qed
```

Lemma sysInit_init_seq shows that the system initialisations can also be expressed by means of the recursive function *map-Upd-seq*.

```
lemma sysInit-init-seq:
  assumes steps: stepsSys bm com sc x y
      and exec-x: systemExec x = Initialize (scheduleInit sc)
      and exec-y: systemExec y = Initialize []
    shows appovar y ∈ map-Upd-seq (systemAppInit bm) {appovar x} (scheduleInit sc)
  using assms by (simp add: sysInit-seq)
```

Lemma sysInit_init_merge shows that the order of initialisations does not matter.

```
lemma sysInit-init-merge:
  assumes wf-bm: wf-BehaviorModel bm
      and wf-sch: wf-SystemSchedule (bmmodel bm) sc
      and steps: stepsSys bm com sc x y
      and exec-x: systemExec x = Initialize (scheduleInit sc)
      and exec-y: systemExec y = Initialize []
    shows appovar y ∈
    {appovar x} ** {⨄_m set ms |ms. map-seq-in ms (map (maps-of (systemAppInit bm)) (scheduleInit sc))}
proof −
  have h0: isInitializing x by (simp add: exec-x)
  have h1: card (set (scheduleInit sc)) = length (scheduleInit sc)
    using wf-sch unfolding wf-SystemSchedule-def by simp
  have h2: ∀ c ∈ modelCIDs (bmmodel bm). (∃ ws qs. appInit (bmbehaviors bm $ c) ws qs)
   using wf-bm unfolding wf-BehaviorModel-def wf-AppBehaviors-def wf-AppBehavior-def wf-InitBehavior-Inhabited-def
by auto
  hence h3: ∀ x ∈ set (scheduleInit sc). systemAppInit bm x ≠ {}
    using wf-sch unfolding wf-SystemSchedule-def systemAppInit-def by simp
  show ?thesis
    using exec-x exec-y h0 steps h1 h3
      sysInit-init-seq[of bm com sc x y]
      map-Upd-Merge[of (scheduleInit sc) (systemAppInit bm) {appovar x}]
```

by *blast*
qed

Lemma sysInit_init_prop describes that after initialisation in any well-formed order all initialisation properties hold, given that the verification properties vc hold. No assumption is made concerning the executed initialisations except that all of them must have been executed. This can also be seen as a consequence of lemma sysInit_init_merge above that shows that any order of initialisations can be replaced by the simultaneous execution of all initialisations.

The proof uses the following strategy. Given the sequence of app initialisations *scheduleInit sc*, executing them with *stepsSys ... y*, the property *sysStateProp ... P y*. In order to show this, we first show that the effect of *stepsSys ... y* can be simulated by *appovar y ∈ map-Upd-seq ...* . This is done by induction, replacing *scheduleInit sc* by a variable *xs*, in lemma sysInit_seq. The term *appovar y ∈ map-Upd-seq ...* can be shown to equal *appovar y ∈ {⨄$_m$ set ms |ms. map-seq-in ms ... (scheduleInit sc))}* using lemmas map_Upd_Merge and map_seq_merge_eq. This establishes that all initial states hold simultaneously. Quantifying over the components *c* and using vc, we prove *sysStateProp ... P y*.

lemma *sysInit-state-prop*:
assumes *wf-bm*: *wf-BehaviorModel bm*
and *wf-sch*: *wf-SystemSchedule (bmmodel bm) sc*
and *wf-state*: *wf-SystemState (bmmodel bm) x*
and *exec-x*: *systemExec x = Initialize (scheduleInit sc)*
and *exec-y*: *systemExec y = Initialize []*
and *steps*: *stepsSys bm com sc x y*
and *vc*: ⋀*c*. *c ∈ modelCIDs (bmmodel bm) ⟹ appInitProp (bmbehaviors bm $ c) (P c)*
shows *sysStateProp bm P y*
proof −
have *i0*: *isInitializing x* **by** (*simp add*: *exec-x*)
have *h0*: *dom (appovar x) ⊆ modelCIDs (bmmodel bm)*
using *wf-state* **unfolding** *appovar-def dom-def wf-SystemState-def* **by** *auto*
have *h1*: *appovar y ∈ {appovar x} ⋆⋆ {⨄$_m$ set ms |ms. map-seq-in ms (map (maps-of (systemAppInit bm)) (scheduleInit sc))}*
using *exec-x exec-y i0 steps sysInit-init-merge wf-bm wf-sch* **by** *blast*
have *g0*: ∀ *m ∈ {⨄$_m$ set ms |ms. map-seq-in ms (map (maps-of (systemAppInit bm)) (scheduleInit sc))}. dom (appovar x) ⊆ dom m*
using *h0 wf-sch* **unfolding** *wf-SystemSchedule-def*
by (*smt* (*verit*) *CollectD map-seq-merge-eq modelCIDs.simps*)
have *g1*: *appovar y ∈ {⨄$_m$ set ms |ms. map-seq-in ms (map (maps-of (systemAppInit bm)) (scheduleInit sc))}*
by (*metis* (*no-types*, *lifting*) *g0 h1 map-Add-extend singleton-iff*)
hence *h2*: ⋀*c*. *c ∈ modelCIDs (bmmodel bm) ⟹ appovar y $ c ∈ systemAppInit bm c*
proof −
fix *c*
assume *a1*: *c ∈ modelCIDs (bmmodel bm)*
have *card (set (scheduleInit sc)) = length (scheduleInit sc)*
using *wf-SystemSchedule-def wf-sch* **by** *presburger*
show *appovar y $ c ∈ systemAppInit bm c*
using *a1 g1 map-seq-merge-el wf-SystemSchedule-def wf-sch* **by** *fastforce*
qed
hence *h3*: ∀ *c ∈ modelCIDs (bmmodel bm).*
(appo (systemThread y $ c), tvar (systemThread y $ c)) ∈ systemAppInit bm c

```
    using g1 wf-sch wf-state unfolding wf-SystemSchedule-def
    by (smt (verit, best) CollectD domIff map-seq-merge-eq map-some-val-given modelCIDs.simps appovar-def)
  have h4: ∀ c ∈ modelCIDs (bmmodel bm). P c (appo (systemThread y $ c), tvar (systemThread y $ c))
    by (smt (verit, best) appInitProp-def h3 mem-Collect-eq systemAppInit-def vc)
  thus ?thesis using sysStateProp-def by blast
qed

definition sysAppInvProp :: CompIds
                          ⇒ (CompId ⇒ ('a PortState × 'a VarState ⇒ bool))
                          ⇒ (CompId ⇒ 'a VarState ⇒ bool)
                          ⇒ bool
  where sysAppInvProp cids P I ≡ ∀ c ∈ cids. ∀ ao tvo. P c (ao, tvo) ⟶ I c tvo




definition appInvProp ::
    'a AppBehavior ⇒
    ('a VarState ⇒ bool) ⇒  — Invariant on variable states
    (('a PortState × 'a VarState) ⇒ bool) ⇒
    bool
  where appInvProp a I P ≡
  ∀ ai tvi d ao tvo. (I tvi) ∧ ((appCompute a) ai tvi d ao tvo) ⟶ (I tvo) ∧ P (ao, tvo)

definition sysInvProp ::
  'a BehaviorModel
     ⇒ ScheduleState × ScheduleState
       ⇒ (CompId ⇒ 'a VarState ⇒ bool)
         ⇒ (CompId ⇒ 'a PortState × 'a VarState ⇒ bool)
           ⇒ bool
  where sysInvProp bm sc I P ≡
  ∀ c ∈  modelCIDs (bmmodel bm).
    ∀ (t:: 'a ThreadState) t'. I c (tvar t) ∧
      stepThread (computeDispatchStatus (bmmodel bm) c)
                 (kindPID (bmmodel bm))
                 ((bmbehaviors bm) $ c)
                 sc
                 t t'
      ⟶ I c (tvar t) ∧ P c (appo t, tvar t)

lemma ⟦ (stepSys am cm sc)** s s'; isInitializing s' ⟧ ⟹ (initStepSys am cm sc)** s s'
proof (induction rule: rtranclp.induct)
  case (rtrancl-refl a)
  then show ?case by simp
next
  case (rtrancl-into-rtrancl a b c)
  have h: isInitializing b
    using rtrancl-into-rtrancl.hyps(2) rtrancl-into-rtrancl.prems stepSys-initializing-back by blast
  hence (initStepSys am cm sc)** a b using rtrancl-into-rtrancl.IH by blast
  then show ?case by (meson h rtrancl-into-rtrancl.hyps(2) rtranclp.simps stepSys-initializing)
```

qed

lemma ⟦ $(stepSys\ am\ cm\ sc)^{**}\ s\ s'$; $isComputing\ s$ ⟧ ⟹ $(computeStepSys\ am\ cm\ sc)^{**}\ s\ s'$
proof (*induction rule*: *rtranclp.induct*)
case (*rtrancl-refl a*)
then show *?case* **by** *blast*
next
case (*rtrancl-into-rtrancl a b c*)
have *h*: $(computeStepSys\ am\ cm\ sc)^{**}\ a\ b$
using *rtrancl-into-rtrancl.IH rtrancl-into-rtrancl.prems* **by** *blast*
have *isComputing b*
using *rtrancl-into-rtrancl.hyps*(*1*) *rtrancl-into-rtrancl.prems*
apply *simp* **by** (*metis Exec.distinct*(*1*) *Exec.exhaust sysInit-seq-bw-supseq*)
then show *?case* **using** *initialize-not-compute*
by (*meson h rtrancl-into-rtrancl.hyps*(*2*) *rtranclp.rtrancl-into-rtrancl stepSys-def*)
qed

lemma *stepSysDcmpL*: $(stepSys\ am\ cm\ sc)^{**}\ s\ s' \Longrightarrow ((initStepSys\ am\ cm\ sc)^{**}\ OO\ (computeStepSys\ am\ cm\ sc)^{**})\ s\ s'$
proof (*induction rule*: *rtranclp.induct*)
case (*rtrancl-refl a*)
then show *?case* **by** *blast*
next
case (*rtrancl-into-rtrancl a b c*)
obtain *x* **where** *x1*: $(initStepSys\ am\ cm\ sc)^{**}\ a\ x$ **and** *x2*: $(computeStepSys\ am\ cm\ sc)^{**}\ x\ b$
using *rtrancl-into-rtrancl.IH* **by** *blast*
then show *?case*
proof (*cases x = b*)
case *True*
then show *?thesis* **by** (*metis relcomppI rtrancl-into-rtrancl.hyps*(*2*) *rtranclp.simps stepSys-def x1*)
next
case *False*
have *isComputing x* **using** *False compute-not-initialize*
by (*metis Exec.exhaust converse-rtranclpE init-init isComputing.elims*(*3*) *x2*)
hence *computeStepSys am cm sc b c*
using *stepSys-def stepSys-initializing-back x2 compute-not-initialize initialize-not-compute*
by (*metis Exec.exhaust isComputing.simps isInitializing.elims*(*3*) *rtrancl-into-rtrancl.hyps*(*2*) *rtranclp.cases*)
then show *?thesis* **using** *x1 x2* **by** *auto*
qed
qed

lemma *stepSysDcmpR*:
assumes $((initStepSys\ am\ cm\ sc)^{**}\ OO\ (computeStepSys\ am\ cm\ sc)^{**})\ s\ s'$
shows $(stepSys\ am\ cm\ sc)^{**}\ s\ s'$
proof −
have $((stepSys\ am\ cm\ sc)^{**}\ OO\ (stepSys\ am\ cm\ sc)^{**})\ s\ s'$
using *stepSys-init-rtranclp stepSys-compute-rtranclp* **using** *assms* **by** *blast*
thus *?thesis* **by** *force*
qed

lemma *stepSysDcmp*: $(stepSys\ am\ cm\ sc)^{**} = (initStepSys\ am\ cm\ sc)^{**}\ OO\ (computeStepSys\ am\ cm\ sc)^{**}$
proof −
have $\bigwedge s\ s'.\ (stepSys\ am\ cm\ sc)^{**}\ s\ s' = ((initStepSys\ am\ cm\ sc)^{**}\ OO\ (computeStepSys\ am\ cm\ sc)^{**})\ s\ s'$
using *stepSysDcmpL stepSysDcmpR* **by** *metis*
thus *?thesis* **by** *presburger*
qed

lemma *sysBehaviorDcmp*:
assumes *wf-bm*: *wf-BehaviorModel bm*
and *wf-sch*: *wf-SystemSchedule* (*bmmodel bm*) *sc*
and *init*: *initSys bm sc x*
and *steps*: *stepsSys bm cm sc x z*
and *comp*: *isComputing z*
and *vc*: $\bigwedge c.\ c \in modelCIDs\ (bmmodel\ bm) \Longrightarrow appInitProp\ (bmbehaviors\ bm\ \$\ c)\ (P\ c)$
shows $\exists y.\ initStepsSys\ bm\ cm\ sc\ x\ y \wedge sysStateProp\ bm\ P\ y \wedge computeStepsSys\ bm\ cm\ sc\ y\ z$
proof −
obtain *y* **where** *y1*: $(initStepSys\ bm\ cm\ sc)^{**}\ x\ y$ **and** *y2*: $(computeStepSys\ bm\ cm\ sc)^{**}\ y\ z$
by (*metis pick-middlep stepSysDcmpL steps stepsSys-def*)
have *h1*: *isInitializing x* **using** *init initSys-def* **by** *auto*
have *h2*: *isComputing y* **using** *y2 comp compute-compute compute-not-initialize*
by (*metis Exec.exhaust converse-rtranclpE isInitializing.simps*)
have $x \neq y$ **using** *compute-not-init h1 h2* **by** *blast*
then obtain y' **where** *y3*: $(initStepSys\ bm\ cm\ sc)^{**}\ x\ y'$ **and** *y4*: $initStepSys\ bm\ cm\ sc\ y'\ y$
by (*metis rtranclp.cases y1*)
have *h3*: $systemExec\ y' = Initialize\ []$
by (*meson compute-not-init h2 initStepSys.cases stepSysInit-initInv-ruleinv stepSys-init y4*)
have *h4*: $systemExec\ x = Initialize\ (scheduleInit\ sc)$ **using** *init initSys-def* **by** *blast*
have *h5*: *wf-SystemState* (*bmmodel bm*) *x* **using** *init initSys-def* **by** *blast*
have $sysStateProp\ bm\ P\ y'$
using *sysInit-state-prop*[*of bm sc x* y' *cm P*] **using** *wf-bm wf-sch* **using** *h3 h4 vc y3 h5*
by (*metis stepSys-init-rtranclp stepsSys-def*)
hence *sysStateProp bm P y*
using *y4 h2 stepSys-init*[*of bm cm sc* y' *y*] **unfolding** *sysStateProp-def* **apply** *clarify*
using *initStepSys.simps stepSysInit-initInv-ruleinv* **by** *fastforce*
then show *?thesis* **by** (*metis computeStepsSys-def initStepsSys-def y1 y2*)
qed

end

Part III

Libraries

Chapter 7

State Reordering

This chapter describes states as partial maps and defines concepts that permit reasoning and reordering of sequences of states and state updates.

theory *SetsAndMaps*
  imports *Main*
begin

definition *opt-get* :: *'a option* ⇒ *'a*
  where [*simp add*]: *opt-get optval* ≡ (*THE v . optval* = *Some v*)

definition *map-get* :: (*'a*, *'b*) *map* ⇒ *'a* ⇒ *'b* (**infixl** $ *73*)
  where [*simp add*]: *map-get m k* = *opt-get* (*m k*)

lemma *map-some-val*:
  assumes *x* ∈ *dom f*
  shows (*f x* = *Some y*) = (*f* $ *x* = *y*)
proof
  show *f x* = *Some y* ⟹ *f* $ *x* = *y*
    by *simp*
next
  show *f* $ *x* = *y* ⟹ *f x* = *Some y*
    using *assms* **by** *force*
qed

lemma *map-some-val-given*:
  assumes *f x* = *Some y*
  shows *f* $ *x* = *y*
  by (*simp add*: *assms*)

lemma *singleton-unfold*: [*a* ↦ *b*] = (λ*x*. *if x* = *a then Some b else None*)
  by *fastforce*

lemma *singleton-map-upd*: *m*(*a*↦*b*) = *m* ++ [*a* ↦ *b*]
  unfolding *map-add-def singleton-unfold*

by *fastforce*

definition *map-Add* (**infixl** $**$ *100*) **where** *map-Add X Y* = $\{x +\!\!+ y \mid x\ y.\ x \in X \wedge y \in Y\}$

lemma *map-Add-assoc*: $X ** (Y ** Z) = (X ** Y) ** Z$
proof
show $X ** Y ** Z \subseteq X ** (Y ** Z)$
proof
fix x
assume $x \in X ** Y ** Z$
then show $x \in X ** (Y ** Z)$ **unfolding** *map-Add-def* **apply** *simp*
by (*metis map-add-assoc*)
qed
next
show $X ** (Y ** Z) \subseteq X ** Y ** Z$ **unfolding** *map-Add-def* **apply** *simp* **by** *force*
qed

lemma *map-Add-unit-right*[*simp*]: $X ** \{Map.empty\} = X$ **unfolding** *map-Add-def* **by** *force*
lemma *map-Add-unit-left*[*simp*]: $\{Map.empty\} ** X = X$ **unfolding** *map-Add-def* **by** *force*
lemma *map-Add-empty-right*[*simp*]: $X ** \{\} = \{\}$ **unfolding** *map-Add-def* **by** *simp*
lemma *map-Add-empty-left*[*simp*]: $\{\} ** X = \{\}$ **unfolding** *map-Add-def* **by** *simp*

lemma *map-Add-extract*: $\{s +\!\!+ m \mid s\ m.\ s \in S \wedge p\ m\} = S ** \{\ m \mid m.\ p\ m\ \}$
proof
show $\{s +\!\!+ m \mid s\ m.\ s \in S \wedge p\ m\} \subseteq S ** \{m \mid m.\ p\ m\}$ **using** *map-Add-def* **by** *fastforce*
next
show $S ** \{m \mid m.\ p\ m\} \subseteq \{s +\!\!+ m \mid s\ m.\ s \in S \wedge p\ m\}$ **by** (*simp add*: *map-Add-def*)
qed

lemma *map-Add-over*:
assumes *x*: $x \in S ** T$
and *d*: $\forall y \in T.\ dom\ x \subseteq dom\ y$
shows $x \in T$
proof −
obtain *p q* **where** *x1*: $p \in S$ **and** *x2*: $q \in T$ **and** *x3*: $x = p +\!\!+ q$ **using** *x*
by (*smt* (*verit*) *CollectD map-Add-def*)
then have *h1*: $dom\ x \subseteq dom\ q$ **using** *d* **by** *simp*
then show *?thesis*
by (*metis Un-iff dom-map-add map-add-subsumed1 map-add-subsumed2 map-le-def map-le-map-add*
subsetI subset-antisym x2 x3)
qed

lemma *map-Add-extend*:
assumes *x*: $x \in S ** T$
and *d*: $\forall a \in S. \forall b \in T.\ dom\ a \subseteq dom\ b$
shows $x \in T$
using *x d*
by (*smt* (*verit*) *CollectD Un-absorb2 dom-map-add map-Add-def map-le-antisym map-le-def map-le-map-add*)

definition *merge* :: $('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ (**infixl** $\uplus_m$ *55*) **where**

$m_1 \uplus_m m_2 \equiv \lambda a.$ *if* $\exists y.$ $\{Some\ y\} = \{m_1\ a, m_2\ a\} - \{None\}$ *then* (*THE* $b.$ $b \in \{m_1\ a, m_2\ a\} - \{None\}$) *else None*

lemma *merge-unit*[*simp*]: $m \uplus_m Map.empty = m$
proof
  fix x
  show $(m \uplus_m Map.empty)\ x = m\ x$
    proof (*cases* $m\ x = None$)
      case *True*
      then show *?thesis*
        by (*simp add*: *merge-def*)
    next
      case *False*
      then show *?thesis*
        apply (*simp add*: *merge-def*)
        by *auto*
    qed
qed

lemma *merge-comm*: $m_1 \uplus_m m_2 = m_2 \uplus_m m_1$
proof
  fix x
  show $(m_1 \uplus_m m_2)\ x = (m_2 \uplus_m m_1)\ x$
  proof (*cases* $(m_1 \uplus_m m_2)\ x = None$)
    case *True*
    then show *?thesis*
      by (*smt* (*verit*) *insertI1 insert-commute merge-def singletonD the-equality*)
  next
    case *False*
    then show *?thesis*
      by (*simp add*: *insert-commute merge-def*)
  qed
qed

lemma *map-merge-left-sub*: $dom\ m_1 \cap dom\ m_2 = \{\} \Longrightarrow m_1 \subseteq_m (m_1 \uplus_m m_2)$
proof (*simp only*: *map-le-def*; *standard*)
  fix a
  assume *a1*: $dom\ m_1 \cap dom\ m_2 = \{\}$
    and *a2*: $a \in dom\ m_1$
  have *x1*: $m_2\ a = None$
    using *a1 a2* **by** *blast*
  from *a2* **obtain** y **where** *y1*: $m_1\ a = Some\ y$ **by** *blast*
  show $m_1\ a = (m_1 \uplus_m m_2)\ a$ **using** *a1* **unfolding** *merge-def dom-def* **apply** (*simp add*: *x1 y1*)
    by (*smt* (*verit*, *best*) *Diff-insert-absorb emptyE insertE insert-commute option.distinct*(1) *the-equality*)
qed

lemma *map-merge-right-sub*: $dom\ m_1 \cap dom\ m_2 = \{\} \Longrightarrow m_2 \subseteq_m (m_1 \uplus_m m_2)$
  by (*metis inf-commute map-merge-left-sub merge-comm*)

```
lemma assoc-disjoint:
  assumes d1: dom m₁ ∩ dom m₂ = {}
      and d2: dom m₁ ∩ dom m₃ = {}
      and d3: dom m₂ ∩ dom m₃ = {}
    shows (m₁ ⊎ₘ m₂) ⊎ₘ m₃ = m₁ ⊎ₘ (m₂ ⊎ₘ m₃)
proof −
  have h1: (m₁ ⊎ₘ m₂) ⊎ₘ m₃ ⊆ₘ m₁ ⊎ₘ (m₂ ⊎ₘ m₃)
  proof (simp only: map-le-def; standard)
    fix a
    assume a ∈ dom (m₁ ⊎ₘ m₂ ⊎ₘ m₃)
    then obtain y where y1: (m₁ ⊎ₘ m₂ ⊎ₘ m₃) a = Some y by blast
    hence h4: Some y = (m₁ ⊎ₘ (m₂ ⊎ₘ m₃)) a
    proof (cases a ∈ dom m₁)
      case True
      have u1: m₁ a = Some y
        using y1 True d1 apply (simp add: merge-def)
        by (smt (z3) Diff-iff d2 domIff insert-absorb insert-disjoint(2) insert-iff option.distinct(1) the1-equality)
      then show ?thesis
        using True d1 d2 apply (simp add: merge-def)
      by (smt (verit) Diff-insert-absorb domIff insertE insert-absorb insert-absorb2 insert-commute insert-disjoint(2)
insert-not-empty option.distinct(1) the-equality)
    next
      case False
      have u2: m₁ a = None using False by blast
      then show ?thesis
      proof (cases a ∈ dom m₂)
        case True
        have u1: m₂ a = Some y
          using y1 True d2 apply (simp add: merge-def)
         by (smt (z3) Diff-iff d3 domIff insert-absorb insert-disjoint(1) insert-iff option.distinct(1) the1-equality)
        have u3: m₃ a = None using True d3 by blast
      then show ?thesis
        using True u1 u2 u3 apply (simp add: merge-def)
      by (smt (verit, best) Diff-cancel empty-Diff insert-Diff-if member-remove not-None-eq remove-def the-equality)
      next
        case False
        have u3: m₂ a = None using False by blast
        have u1: m₃ a = Some y
          using y1 u2 u3 apply (simp add: merge-def)
          by (smt (z3) Diff-cancel Diff-insert-absorb insert-absorb2 insert-not-empty option.simps(3) singletonD
theI-unique)
        then show ?thesis using False u1 u2 u3 by (simp add: merge-def)
      qed
    qed
    show (m₁ ⊎ₘ m₂ ⊎ₘ m₃) a = (m₁ ⊎ₘ (m₂ ⊎ₘ m₃)) a by (simp add: h4 y1)
  qed
  have h2: m₁ ⊎ₘ (m₂ ⊎ₘ m₃) ⊆ₘ (m₁ ⊎ₘ m₂) ⊎ₘ m₃
  proof (simp only: map-le-def; standard)
    fix a
```

```
    assume a1: a ∈ dom (m₁ ⊎ₘ (m₂ ⊎ₘ m₃))
    then obtain y where y1: (m₁ ⊎ₘ (m₂ ⊎ₘ m₃)) a = Some y by blast
    hence h5: Some y = (m₁ ⊎ₘ m₂ ⊎ₘ m₃) a
    proof (cases a ∈ dom m₁)
      case True
      have u1: m₁ a = Some y
        using y1 True apply (simp add: merge-def)
        by (smt (z3) Diff-iff a1 domIff insertCI insertE the-equality y1)
      have u2: m₂ a = None using True d1 by blast
      have u3: m₃ a = None using True d2 by blast
      then show ?thesis
        using True u1 u2 u3 y1 apply (simp add: merge-def)
        by (smt (verit) option.distinct(1) the-equality)
    next
      case False
      have u2: m₁ a = None using False by blast
      then show ?thesis
      proof (cases a ∈ dom m₂)
        case True
        have u1: m₂ a = Some y
          using y1 u2 True d2 apply (simp add: merge-def)
          by (smt (z3) Diff-iff domIff insert-iff option.simps(3) theI)
        have u3: m₃ a = None using True d3 by blast
        then show ?thesis
          using u1 u2 u3 apply (simp add: merge-def)
          by (smt (verit, best) Diff-insert-absorb emptyE insertE insert-commute option.distinct(1) the-equality)
      next
        case False
        have u1: m₂ a = None using False by blast
        have u3: m₃ a = Some y
          using y1 u2 u1 apply (simp add: merge-def)
          by (smt (z3) Diff-iff insert-Diff1 option.distinct(1) singletonD singletonI theI)
        then show ?thesis using False u1 u2 u3 by (simp add: merge-def)
      qed
    qed
    show (m₁ ⊎ₘ (m₂ ⊎ₘ m₃)) a = (m₁ ⊎ₘ m₂ ⊎ₘ m₃) a by (simp add: h5 y1)
  qed
  show ?thesis using h1 h2 map-le-antisym by blast
qed

definition Merge :: ('a ⇀ 'b) set ⇒ ('a ⇀ 'b)  (⨄ₘ - [900] 900) where
  (⨄ₘ M) ≡ λa. if (∃ y. {Some y} = { b | b m. m ∈ M ∧ m a = b  ∧ b ≠ None })
    then (THE b. ∃  m. m ∈ M ∧ m a = b ∧ b ≠ None)
    else None

lemma Merge-empty[simp]: (⨄ₘ {}) = Map.empty
  unfolding Merge-def by simp

lemma map-Merge-not-none: (⨄ₘ M) a ≠ None ⟹ ∃ y. ∀ m ∈ M. m a ≠ None ⟶ m a = Some y
proof −
```

```
  assume a1: (⨄ m M) a ≠ None
  then obtain y where y1: (⨄ m M) a = Some y by blast
  hence h1: Some y = (THE b. ∃ m. m ∈ M ∧ m a = b ∧ b ≠ None)
    by (metis (mono-tags, lifting) Merge-def option.distinct(1))
  have ∀ m ∈ M. m a ≠ None ⟶ m a = Some y
  proof
    fix m
    assume m1: m ∈ M
    show m a ≠ None ⟶ m a = Some y
    proof
      assume m2: m a ≠ None
      show m a = Some y using a1 m1 m2 apply(simp add: h1 Merge-def)
        by (smt (verit, del-insts) mem-Collect-eq option.distinct(1) singleton-iff theI-unique)
    qed
  qed
  thus ?thesis by blast
qed

lemma map-Merge-le: ⟦m ∈ M; ∀ m' ∈ M. dom m ∩ dom m' = {}⟧ ⟹ (⨄ m M) |` dom m ⊆m m
  by fastforce

lemma map-Merge-dom-sub: dom (⨄ m M) ⊆ (⋃ m ∈ M. dom m)
proof
  fix x
  assume a1: x ∈ dom (⨄ m M)
  then show x ∈ (⋃ m ∈ M. dom m)
    unfolding dom-def apply(simp add: Merge-def)
    by (smt (verit, best) insertI1 mem-Collect-eq option.distinct(1))
qed

lemma map-Merge-dom:
  assumes ∀ m1 ∈ M.∀ m2 ∈ M. dom m1 ∩ dom m2 = {}
  shows dom (⨄ m M) = (⋃ m ∈ M. dom m)
proof
  show dom (⨄ m M) ⊆ (⋃ m ∈ M. dom m)
  proof
    fix x
    assume x ∈ dom (⨄ m M)
    then show x ∈ (⋃ m ∈ M. dom m)
      unfolding dom-def apply(simp add: Merge-def)
      by (smt (verit) insertI1 mem-Collect-eq option.distinct(1))
  qed
next
  show (⋃ m ∈ M. dom m) ⊆ dom (⨄ m M)
  proof
    fix x
    assume x ∈ (⋃ m ∈ M. dom m)
    then show x ∈ dom (⨄ m M)
      unfolding dom-def apply(simp add: Merge-def)
      using assms by fastforce
```

```
  qed
qed

lemma not-dom-Merge:
  assumes a: a ∉ M
      and d: ∀ m ∈ M. dom a ∩ dom m = {}
    shows dom a ∩ dom (⨄ m M) = {}
proof
  have ⋀x. x ∈ dom a ⟹ x ∉ dom (⨄ m M)
    using a d unfolding Merge-def apply (simp add: dom-def; clarify)
    by (smt (verit) domI empty-Collect-eq insert-disjoint(2) insert-dom insert-not-empty)
  thus dom a ∩ dom (⨄ m M) ⊆ {} by blast
next
  show {} ⊆ dom a ∩ dom (⨄ m M) by blast
qed

lemma empty-Merge:
  assumes a: a = Map.empty
    shows dom a ∩ dom (⨄ m M) = {}
  by (simp add: assms)

lemma merge-Merge-union:
  assumes d: ∀ m' ∈ M. dom m ∩ dom m' = {}
  shows ⨄ m (M∪{m}) = ⨄ m M ⊎m m
proof (rule map-le-antisym)
  show h1: (⨄ m (M ∪ {m})) ⊆m (⨄ m M ⊎m m)
  proof (simp only: map-le-def; standard)
    fix a
    assume a1: a ∈ dom (⨄ m (M ∪ {m}))
    then obtain y where a2: Some y = (⨄ m (M ∪ {m})) a by force
    have Some y = (⨄ m M ⊎m m) a
    proof (cases a ∈ dom m)
      case True
      then have t1: a ∈ dom m by blast
      then have a3: Some y = m a using a2 unfolding Merge-def
        by (metis (mono-tags, lifting) UnI2 domIff mem-Collect-eq option.simps(3)
            singletonD singletonI the-equality)
      show ?thesis
      proof (cases a ∈ dom (⨄ m M))
        case True
        have (⨄ m M) a = Some y
          using t1 True d unfolding Merge-def apply (simp only: a2 a3)
          by (smt (verit) domIff empty-Collect-eq insert-absorb insert-disjoint(2) insert-not-empty)
        then show ?thesis using t1 d a3 unfolding Merge-def
          by (smt (verit, best) domIff insert-disjoint(1) insert-dom mem-Collect-eq singletonI)
      next
        case False
        then have t2: (⨄ m M) a = None by (simp add: domIff)
        show ?thesis using False t1 d a3 unfolding merge-def apply (simp add: t2)
          by (smt (verit, ccfv-threshold) Diff-empty Diff-insert0 emptyE insertE the-equality)
```

```
    qed
  next
    case False
    hence t4: m a = None by blast
    hence t5: ∀ m' ∈ M. m' a = Some y ∨ m' a = None
      using a2 t4 unfolding Merge-def
      by (metis (mono-tags, lifting) Un-iff mem-Collect-eq option.simps(3) singleton-iff the-equality)
    then obtain m' where t6: m' ∈ M and t7: m' a = Some y
      using a1 unfolding Merge-def
      by (smt (verit, ccfv-threshold) False Un-iff domIff empty-Collect-eq insert-not-empty singletonD)
    hence t5: (⨄ m M) a = Some y using t5 unfolding Merge-def apply simp
      by (smt (verit, ccfv-threshold) Collect-cong option.distinct(1) singleton-conv the-equality)
    then show ?thesis using t5 unfolding merge-def
      by (simp add: insert-commute t4)
  qed
  then show (⨄ m (M ∪ {m})) a = (⨄ m M ⊎m m) a by (simp add: a2)
 qed
next
 show h2: (⨄ m M ⊎m m) ⊆m (⨄ m (M ∪ {m}))
 proof (simp only: map-le-def; standard)
  fix a
  assume a3: a ∈ dom (⨄ m M ⊎m m)
  then obtain y where b1: Some y = (⨄ m M ⊎m m) a by force
  have b2: Some y = (⨄ m (M ∪ {m})) a
  proof (cases a ∈ dom m)
    case True
    hence b3: m a = Some y using b1 unfolding merge-def
      by (smt (z3) Diff-iff domIff empty-Collect-eq insert-Diff1 insert-Diff-single insert-absorb
          insert-commute insert-iff insert-not-empty merge-def not-Some-eq singletonD singletonI theI)
    hence b5: ∀ m' ∈ M. m' a = Some y ∨ m' a = None
      using True d by blast
    hence b6: ∀ m' ∈ M∪{m}. m' a = Some y ∨ m' a = None using b3 by fastforce
    then show ?thesis unfolding Merge-def
      by (smt (z3) Collect-cong True Un-insert-right domIff insert-iff singleton-conv2 the-equality)
  next
    case False
    hence b7: m a = None by blast
    hence b8: Some y = (⨄ m M) a using b1 unfolding merge-def
      by (metis Diff-iff insertI1 insert-absorb2 option.distinct(1) singletonD the-equality)
    hence b9: ∀ m' ∈ M. m' a = Some y ∨ m' a = None unfolding Merge-def
      by (metis (mono-tags, lifting) mem-Collect-eq option.simps(3) singleton-iff the-equality)
    hence t8: ∀ m' ∈ M∪{m}. m' a = Some y ∨ m' a = None using b7 by simp
    from b8 b9 obtain m' where x1: m' ∈ M and x2: m' a = Some y unfolding Merge-def
      by (smt (verit, best) all-not-in-conv empty-not-insert mem-Collect-eq option.distinct(1))
    then show ?thesis using t8 unfolding Merge-def
      by (smt (z3) Collect-cong Collect-empty-eq UnCI singleton-conv the-equality)
  qed
  show (⨄ m M ⊎m m) a = (⨄ m (M ∪ {m})) a
    using b1 b2 by presburger
 qed
```

qed

lemma *merge-Merge-diff*:
assumes *m*: $m \in M$
and *d*: $\forall m' \in M - \{m\}.\ dom\ m \cap dom\ m' = \{\}$
shows $\biguplus_m\ M = \biguplus_m\ (M - \{m\}) \uplus_m m$
using *d m merge-Merge-union*[*of* $M - \{m\}$ *m*] **by** (*simp add*: *insert-absorb*)

lemma *map-add-merge*:
assumes *d*: $dom\ m_1 \cap dom\ m_2 = \{\}$
shows $m_1 ++ m_2 = m_1 \uplus_m m_2$
proof (*rule map-le-antisym*)
show $m_1 ++ m_2 \subseteq_m m_1 \uplus_m m_2$
proof (*simp only*: *map-le-def*; *standard*)
fix *a*
assume *a1*: $a \in dom\ (m_1 ++ m_2)$
then obtain *y* **where** *y1*: $Some\ y = (m_1 ++ m_2)\ a$ **by** (*metis domD*)
hence $Some\ y = (m_1 \uplus_m m_2)\ a$
proof (*cases* $a \in dom\ m_1$)
case *True*
hence *b0*: $m_2\ a = None$ **using** *d* **by** *blast*
hence *b1*: $(m_1 ++ m_2)\ a = m_1\ a$ **using** *d* **by** (*simp add*: *domIff map-add-dom-app-simps*(*3*))
then show *?thesis* **using** *b0 y1* **unfolding** *merge-def* **apply** (*simp add*: *b1*)
by (*smt* (*verit*, *ccfv-SIG*) *Diff-insert-absorb empty-iff insertE insert-commute*
option.distinct(*1*) *the-equality*)
next
case *False*
hence *b2*: $(m_1 ++ m_2)\ a = m_2\ a$ **using** *d* **by** (*simp add*: *map-add-dom-app-simps*(*2*))
then show *?thesis* **using** *False y1* **unfolding** *merge-def* **apply** (*simp add*: *b2*)
by (*smt* (*verit*, *ccfv-threshold*) *Diff-insert-absorb domIff empty-iff insert-iff*
option.distinct(*1*) *the-equality*)
qed
then show $(m_1 ++ m_2)\ a = (m_1 \uplus_m m_2)\ a$ **using** *y1* **by** *presburger*
qed
next
show $m_1 \uplus_m m_2 \subseteq_m m_1 ++ m_2$
proof (*simp only*: *map-le-def*; *standard*)
fix *a*
assume *a2*: $a \in dom\ (m_1 \uplus_m m_2)$
then obtain *y* **where** *y1*: $Some\ y = (m_1 \uplus_m m_2)\ a$ **by** (*metis domD*)
hence $Some\ y = (m_1 ++ m_2)\ a$
proof (*cases* $a \in dom\ m_1$)
case *True*
hence *b3*: $m_2\ a = None$ **using** *d* **by** *blast*
have *b4*: $m_1\ a = Some\ y$ **using** *y1 True d* **unfolding** *merge-def* **apply** (*simp only*: *b3*)
by (*metis* (*no-types*, *lifting*) *Diff-iff insertE option.distinct*(*1*) *singletonI the-equality*)
then show *?thesis* **by** (*simp add*: *b3 map-add-def*)
next
case *False*
hence *b5*: $m_1\ a = None$ **by** *blast*

```
    have b6: m2 a = Some y using y1 False d unfolding merge-def apply (simp only: b5)
     by (metis Diff-iff insertCI insertE not-None-eq theI)
    then show ?thesis by simp
   qed
   then show (m1 ⊎m m2) a = (m1 ++ m2) a using y1 by presburger
  qed
qed

fun map-add-seq where
  map-add-seq s [] = s
| map-add-seq s (m#ms) = map-add-seq (s ++ m) ms

lemma map-add-seq-foldl: map-add-seq s ms = foldl map-add s ms
proof (induction ms arbitrary: s)
  case Nil
  then show ?case by fastforce
next
  case (Cons a ms)
  then show ?case by simp
qed


lemma seq-set-dom:
  ∀ i j. i < length ms ∧ j < length ms ∧ dom (ms!i) ∩ dom (ms!j) ≠ {} ⟶ i = j ⟹
  ∀ m1 ∈ set ms.∀ m2 ∈ set ms. m1 ≠ m2 ⟶ dom m1 ∩ dom m2 = {}
proof (induction ms)
  case Nil
  then show ?case by auto
next
  case (Cons a ms)
  then show ?case by (metis in-set-conv-nth)
qed

lemma map-add-seq-Merge:
  ∀ i j. i < length ms ∧ j < length ms ∧ dom (ms!i) ∩ dom (ms!j) ≠ {} ⟶ i = j
  ⟹ map-add-seq s ms = s ++ (⨄m set ms)
proof (induction ms arbitrary: s)
  case Nil
  then show ?case by simp
next
  case (Cons a ms)
  have h0: ∀ k. k < length ms ⟶ ms ! k = (a # ms) ! (Suc k) by simp
  have h1: ∀ i j. i < length ms ∧ j < length ms ∧ dom (ms!i) ∩ dom (ms!j) ≠ {} ⟶ i = j
  proof (clarify)
    fix i j
    assume a1: i < length ms
      and a2: j < length ms
      and a3: dom (ms!i) ∩ dom (ms!j) ≠ {}
    then show i = j
     by (metis Cons.prems(1) Suc-leI h0 le-imp-less-Suc length-Cons old.nat.inject)
```

```
  qed
  have h2: ∀ m ∈ set ms. a ≠ m ⟶ dom a ∩ dom m = {}
    by (meson Cons.prems(1) list.set-intros(1) list.set-intros(2) seq-set-dom)
  have h3: a ≠ Map.empty ⟹ a ∉ set ms
  proof −
    assume a4: a ≠ Map.empty
    hence a5: dom a ≠ {} by simp
    thus a ∉ set ms using h0 Cons.prems(1)
      by (metis Diff-Diff-Int Diff-cancel Diff-empty Suc-less-eq in-set-conv-nth length-Cons
          nat.simps(3) nth-Cons-0 zero-less-Suc)
  qed
  have h3: dom a ∩ dom (⨄m set ms) = {}
    using h2 h3 not-dom-Merge by fastforce
  have h4: map-add-seq (s ++ a) ms = (s ++ a) ++ ⨄m set ms using Cons.IH card-length h1 by blast
  have h5: ... = s ++ (a ++ ⨄m set ms) by simp
  have h6: ... = s ++ (a ⊎m ⨄m set ms) using map-add-merge[of a ⨄m set ms] by (simp add: h3)
  have h7: ... = s ++ (⨄m (set ms ∪ {a}))
    using h2 h3 map-add-merge not-dom-Merge merge-Merge-union[of set ms a]
    by (smt (verit) DiffD1 DiffD2 DiffI Diff-empty Diff-insert-absorb Int-commute
        Int-empty-right Un-empty-right disjoint-iff-not-equal domIff insert-Diff insert-Diff-single
        map-add-dom-app-simps(1) merge-Merge-diff merge-comm mk-disjoint-insert)
  then show ?case by (metis h4 h5 h6 insert-is-Un list.simps(15) map-add-seq.simps(2) sup-commute)
qed

definition map-add-all where map-add-all ms = map-add-seq Map.empty ms

lemma add-all: ∀ i j. i < length ms ∧ j < length ms ∧dom (ms!i) ∩ dom (ms!j) ≠ {} ⟶ i = j
   ⟹ map-add-all ms = ⨄m set ms
  using map-add-seq-Merge[of ms Map.empty]
  by (simp add: map-add-all-def)

lemma add-all-empty[simp]: map-add-all [] = Map.empty
  by (simp add: map-add-all-def)

lemma add-all-left: m ++ map-add-all ms = map-add-all (m#ms)
proof (induction ms arbitrary: m)
  case Nil
  then show ?case by (simp add: map-add-all-def)
next
  case (Cons a ms)
  then show ?case by (metis map-add-all-def map-add-assoc map-add-seq.simps(2))
qed

fun map-Add-seq where
  map-Add-seq S [] = S
| map-Add-seq S (M#Ms) = map-Add-seq (S ** M) Ms

lemma map-Add-seq-foldl: map-Add-seq S Ms = foldl map-Add S Ms
proof (induction Ms arbitrary: S)
  case Nil
```

```
  then show ?case by fastforce
next
  case (Cons a ms)
  then show ?case by simp
qed

fun map-seq-in where
  map-seq-in [] [] = True
| map-seq-in (x#xs) (X#XS) = (x ∈ X ∧ map-seq-in xs XS)
| map-seq-in - - = False

lemma map-seq-in-ex: ⟦ m ∈ set ms; map-seq-in ms Ms ⟧ ⟹ ∃ M ∈ set Ms. m ∈ M
proof (induction Ms arbitrary: ms)
  case Nil
  then show ?case using map-seq-in.elims(2) by force
next
  case (Cons a ms)
  then show ?case by (metis list.set-cases list.set-intros(1) list.set-intros(2) map-seq-in.simps(2))
qed

lemma map-seq-in-length: map-seq-in ms Ms ⟹ length ms = length Ms
proof (induction Ms arbitrary: ms)
  case Nil
  then show ?case using map-seq-in.elims(2) by auto
next
  case (Cons a Ms)
  then show ?case using map-seq-in.elims(2) by force
qed

lemma map-seq-in-index: ⟦map-seq-in ms Ms; i < length Ms⟧ ⟹ ms ! i ∈ Ms ! i
proof (induction Ms arbitrary: ms i)
  case Nil
  then show ?case by simp
next
  case (Cons a Ms)
  have h1: map-seq-in (tl ms) Ms
    by (metis Cons.prems(1) list.exhaust-sel map-seq-in.simps(2) map-seq-in.simps(4))
  show ?case
  proof (cases i = 0)
    case True
    then show ?thesis using Cons.prems(1) map-seq-in.elims(2) by fastforce
  next
    case False
    obtain j where j1: Suc j = i by (metis False old.nat.exhaust)
    hence h2: j < length Ms using Cons.prems(2) by force
    hence h3: (tl ms) ! j ∈ Ms ! j by (simp add: Cons.IH h1)
    then show ?thesis
      by (metis Cons.prems(1) j1 list.exhaust-sel map-seq-in.simps(4) nth-Cons-Suc)
  qed
qed
```

definition *map-seq-of* **where**
map-seq-of ms Ms ≡ *map-seq-in ms Ms* ∧
(∀ *i j*. *i* < *length ms* ∧ *j* < *length ms* ∧ *dom* (*ms*!*i*) ∩ *dom* (*ms*!*j*) ≠ {} ⟶ *i* = *j*) ∧
(∀ *M* ∈ *set Ms*.∀ m_1 ∈ *M*.∀ m_2 ∈ *M*. *dom* m_1 = *dom* m_2)

lemma *map-seq-of-empty*[*simp*]: *map-seq-of ms* [] ⟹ *ms* = []
using *map-seq-in.elims*(*2*) *map-seq-of-def* **by** *blast*

lemma *map-seq-of-wk*: *map-seq-of ms Ms* ⟹ *map-seq-in ms Ms*
using *map-seq-of-def* **by** *blast*

lemma *map-seq-of-in*:
⟦ *map-seq-of ms Ms*;
m_1 ∈ *set ms*;
m_2 ∈ *set ms*;
m_1 ≠ m_2 ⟧ ⟹ ∃ M_1 ∈ *set Ms*.∃ M_2 ∈ *set Ms*. M_1 ≠ M_2 ∧ m_1 ∈ M_1 ∧ m_2 ∈ M_2
proof (*induction Ms arbitrary*: *ms*)
case *Nil*
then show *?case*
unfolding *map-seq-of-def* **by** (*metis empty-iff list.collapse map-seq-in.simps*(*3*) *set-empty*)
next
case (*Cons a Ms*)
hence *h0*: ∀ *k*. *k* ≤ *length* (*tl ms*) ⟶ (*tl ms*) ! *k* = *ms* ! (*Suc k*)
unfolding *map-seq-of-def* **apply** *clarify*
by (*metis equals0D hd-Cons-tl list.set*(*1*) *nth-Cons-Suc*)
then have *h1*: ∀ *i j*. *i* < *length* (*tl ms*) ∧ *j* < *length* (*tl ms*) ∧ *dom* ((*tl ms*)!*i*) ∩ *dom* ((*tl ms*)!*j*) ≠ {} ⟶ *i* = *j*
using *Cons.prems* **unfolding** *map-seq-of-def* **apply** *clarify*
by (*metis Nitpick.size-list-simp*(*2*) *Suc-inject Suc-leI le-imp-less-Suc length-greater-0-conv*
length-pos-if-in-set nth-tl)
have *h2*: ∀ *M*∈*set Ms*. ∀ m_1∈*M*. ∀ m_2∈*M*. *dom* m_1 = *dom* m_2
using *Cons.prems* **unfolding** *map-seq-of-def* **apply** *clarify* **by** (*meson list.set-intros*(*2*))
have *h4*: *map-seq-in* (*tl ms*) *Ms* **using** *Cons.prems* **unfolding** *map-seq-of-def* **apply** *clarify*
by (*metis length-greater-0-conv length-pos-if-in-set list.exhaust-sel map-seq-in.simps*(*2*))
have *h5*: m_1 ≠ m_2 **by** (*simp add*: *Cons.prems*)
then show *?case*
proof (*cases* m_1 ∈ *set* (*tl ms*))
case *True*
hence *h6*: m_1 ∈ *set* (*tl ms*) **by** *simp*
then show *?thesis*
proof (*cases* m_2 ∈ *set* (*tl ms*))
case *True*
hence *h7*: m_2 ∈ *set* (*tl ms*) **by** *simp*
have ∃ M_1∈*set Ms*. ∃ M_2∈*set Ms*. M_1 ≠ M_2 ∧ m_1 ∈ M_1 ∧ m_2 ∈ M_2
using *Cons.IH Cons.prems*(*4*) *h1 h2 h4 h6 h7* **unfolding** *map-seq-of-def*
by (*smt* (*verit*, *best*) *card-length*)
then show *?thesis* **by** (*meson list.set-intros*(*2*))
next
case *False*

```
      hence h7: m₂ ∉ set (tl ms) by simp
      hence x1: m₂ ∈ a
        using Cons.prems(1) Cons.prems(3) list.set-cases
        unfolding map-seq-of-def apply clarify by fastforce
      obtain M where m₁ ∈ M and M ∈ set Ms by (meson h4 h6 map-seq-in-ex)
      then show ?thesis
        using Cons.prems(1) Cons.prems(2) Cons.prems(3) h5 x1
        unfolding map-seq-of-def apply clarify
        by (smt (verit) dom-eq-empty-conv in-set-conv-nth inf.idem map-seq-in-ex)
    qed
  next
    case False
    hence h6: m₁ ∉ set (tl ms) by simp
    hence x1: m₁ ∈ a using Cons.prems(1) Cons.prems(2) unfolding map-seq-of-def apply clarify
      using list.set-cases by fastforce
    then show ?thesis
    proof (cases m₂ ∈ set (tl ms))
      case True
      hence h7: m₂ ∈ set (tl ms) by simp
      then show ?thesis
        using Cons.prems(1) Cons.prems(2) Cons.prems(3) h5 unfolding map-seq-of-def apply clarify
        by (smt (verit, best) dom-eq-empty-conv in-set-conv-nth inf.idem map-seq-in-ex)
    next
      case False
      then show ?thesis
        by (metis Cons.prems(2) Cons.prems(3) h5 h6 list.exhaust-sel set-ConsD tl-Nil)
    qed
  qed
qed

lemma map-Add-seq-as-add-seq:
  map-Add-seq S Ms = {map-add-seq s ms | s ms. s ∈ S ∧ map-seq-in ms Ms}
proof (induction Ms arbitrary: S)
  case Nil
  then show ?case
  proof
    show map-Add-seq S [] ⊆ {map-add-seq s ms | s ms. s ∈ S ∧ map-seq-in ms []}
    proof
      fix x
      assume x ∈ map-Add-seq S []
      then show x ∈ {map-add-seq s ms |s ms. s ∈ S ∧ map-seq-in ms []}
        apply simp
        by (metis map-add-seq.simps(1) map-seq-in.simps(1))
    qed
  next
    show {map-add-seq s ms | s ms. s ∈ S ∧ map-seq-in ms []} ⊆ map-Add-seq S []
    proof
      fix x
      assume x ∈ {map-add-seq s ms |s ms. s ∈ S ∧ map-seq-in ms []}
      then show x ∈ map-Add-seq S []
```

```
      apply simp
      using map-seq-in.elims(1) by force
    qed
  qed
next
  case (Cons A Ms)
  have h1: map-Add-seq (S ∗∗ A) Ms = {map-add-seq s ms | s ms. s ∈ (S ∗∗ A) ∧ map-seq-in ms Ms}
    using local.Cons by blast
  show ?case
  proof
    show map-Add-seq S (A#Ms) ⊆ {map-add-seq s ms | s ms. s ∈ S ∧ map-seq-in ms (A#Ms)}
    proof
      fix x
      assume a1: x ∈ map-Add-seq S (A # Ms)
      hence a2: x ∈ map-Add-seq (S ∗∗ A) Ms by simp
      obtain s a ms where p1: s ∈ S and
                          p2: a ∈ A and
                          p3: map-seq-in ms Ms and
                          p4: x = map-add-seq (s ++ a) ms
        by (smt (verit, ccfv-SIG) a2 h1 map-Add-def mem-Collect-eq)
      have a3: map-seq-in (a#ms) (A#Ms) by (simp add: p2 p3)
      then show x ∈ {map-add-seq s ms |s ms. s ∈ S ∧ map-seq-in ms (A # Ms)}
        by (smt (verit) map-add-seq.simps(2) mem-Collect-eq p1 p4)
    qed
  next
    show {map-add-seq s ms | s ms. s ∈ S ∧ map-seq-in ms (A#Ms)} ⊆ map-Add-seq S (A#Ms)
    proof
      fix x
      assume a4: x ∈ {map-add-seq s ms |s ms. s ∈ S ∧ map-seq-in ms (A # Ms)}
      then obtain s a ms where q1: s ∈ S and
                               q2: a ∈ A and
                               q3: map-seq-in ms Ms and
                               q4: x = map-add-seq s (a#ms)
        by (smt (verit, del-insts) CollectD list.discI list.sel(1) list.sel(3) map-seq-in.elims(2))
      show x ∈ map-Add-seq S (A # Ms)
        using h1 map-Add-def q1 q2 q3 q4 by fastforce
    qed
  qed
qed

definition Dom where Dom X ≡ ⋃x∈X. (dom x)

lemma Dom-single: assumes ∃ y. P y shows Dom { [a ↦ y] | y. P y } = {a}
proof
  show Dom {[a ↦ y] |y. P y} ⊆ {a}
    unfolding Dom-def dom-def apply (simp; clarify) using domIff by fastforce
next
  show {a} ⊆ Dom {[a ↦ y] |y. P y}
    using assms unfolding Dom-def dom-def apply (simp; clarify) by fastforce
qed
```

definition *seq-of-maps* **where**
seq-of-maps Ms ≡
(∀ *i j*. *i* < *length Ms* ∧ *j* < *length Ms* ∧ *Dom* (*Ms*!*i*) ∩ *Dom* (*Ms*!*j*) ≠ {} ⟶ *i* = *j*) ∧
(∀ *M* ∈ *set Ms*.∀ m_1 ∈ *M*.∀ m_2 ∈ *M*. *dom* m_1 = *dom* m_2)

lemma *seq-of-maps-hd*: **assumes** *seq-of-maps* (*M*#*Ms*) **shows** *seq-of-maps Ms*
proof −
have *h0*: ∀ *k*. *k* < *length Ms* ⟶ *Ms* ! *k* = (*M*#*Ms*) ! (*Suc k*)
by *simp*
thus *?thesis* **using** *assms* **unfolding** *seq-of-maps-def* **apply** *clarify*
by (*metis Suc-inject Suc-leI le-imp-less-Suc length-Cons list.set-intros*(*2*))
qed

lemma *maps-seq-of*:
assumes *M1*: *seq-of-maps Ms*
and *M2*: *map-seq-in ms Ms*
shows *map-seq-of ms Ms*
proof −
have *h1*: *map-seq-in ms Ms* **by** (*simp add*: *M2*)
have *h2*: ∀ *i j*. *i* < *length ms* ∧ *j* < *length ms* ∧ *dom* (*ms* ! *i*) ∩ *dom* (*ms* ! *j*) ≠ {} ⟶ *i* = *j*
proof *clarify*
fix *i j*
assume *a1*: *i* < *length ms*
and *a2*: *j* < *length ms*
and *a3*: *dom* (*ms* ! *i*) ∩ *dom* (*ms* ! *j*) ≠ {}
have *k1*: *ms* ! *i* ∈ *Ms* ! *i* **by** (*metis a1 h1 map-seq-in-index map-seq-in-length*)
have *k2*: *ms* ! *j* ∈ *Ms* ! *j* **by** (*metis a2 h1 map-seq-in-index map-seq-in-length*)
obtain *x* **where** *x1*: *x* ∈ *dom* (*ms* ! *i*) **and** *x2*: *x* ∈ *dom* (*ms* ! *j*) **using** *a3* **by** *blast*
have *k3*: *x* ∈ *Dom* (*Ms*!*i*) **using** *k1 x1* **unfolding** *Merge-def Dom-def* **apply** *clarify* **by** *blast*
have *k4*: *x* ∈ *Dom* (*Ms*!*j*) **using** *k2 x2* **unfolding** *Merge-def Dom-def* **apply** *clarify* **by** *blast*
have *k5*: *Dom* (*Ms*!*i*) ∩ *Dom* (*Ms*!*j*) ≠ {} **using** *k3 k4* **by** *blast*
show *i* = *j* **using** *M1* **unfolding** *seq-of-maps-def* **by** (*metis a1 a2 h1 k5 map-seq-in-length*)
qed
have *h3*: ∀ *M*∈*set Ms*. ∀ m_1∈*M*. ∀ m_2∈*M*. *dom* m_1 = *dom* m_2 **by** (*metis M1 seq-of-maps-def*)
have *h4*: *card* (*set Ms*) ≤ *length Ms* **by** (*simp add*: *card-length*)
show *?thesis* **using** *h1 h2 h3 h4* **unfolding** *map-seq-of-def* **by** *blast*
qed

lemma *map-add-shift*: *M* ∗∗ {*map-add-all ms* | *ms*. *map-seq-in ms Ms*} = {*map-add-all ms* | *ms*. *map-seq-in ms* (*M*#*Ms*)}
proof
show *M* ∗∗ {*map-add-all ms* |*ms*. *map-seq-in ms Ms*} ⊆ {*map-add-all ms* |*ms*. *map-seq-in ms* (*M* # *Ms*)}
proof
fix *x*
assume *a1*: *x* ∈ *M* ∗∗ {*map-add-all ms* |*ms*. *map-seq-in ms Ms*}
then obtain *m ms* **where** *m1*: *m* ∈ *M* **and** *m2*: *map-seq-in ms Ms* **and** *m3*: *x* = *m* ++ *map-add-all ms*
unfolding *map-Add-def* **by** *blast*
have *x* = *map-add-all* (*m*#*ms*) **by** (*simp add*: *add-all-left m3*)
then show *x* ∈ {*map-add-all ms* |*ms*. *map-seq-in ms* (*M* # *Ms*)} **using** *m1 m2* **by** *fastforce*

```
  qed
next
  show {map-add-all ms |ms. map-seq-in ms (M # Ms)} ⊆ M ** {map-add-all ms |ms. map-seq-in ms Ms}
  proof
    fix x
    assume a1: x ∈ {map-add-all ms |ms. map-seq-in ms (M # Ms)}
   then obtain m ms where m1: m ∈ M and m2: map-seq-in (m#ms) (M # Ms) and m3: x = map-add-all
(m#ms)
      apply (simp; clarify) by (metis map-seq-in.elims(2) map-seq-in.simps(2) map-seq-in.simps(4))
    have h1: map-seq-in ms Ms using m2 by force
    have h2: x = m ++ map-add-all ms by (simp add: add-all-left m3)
    show x ∈ M ** {map-add-all ms |ms. map-seq-in ms Ms}
      unfolding map-Add-def using h1 h2 m1 by blast
  qed
qed

lemma map-add-seq-of-shift:
  assumes seq-of-maps (M#Ms)
  shows M ** {map-add-all ms | ms. map-seq-of ms Ms} = {map-add-all ms | ms. map-seq-of ms (M#Ms)}
proof
  show M ** {map-add-all ms |ms. map-seq-of ms Ms} ⊆ {map-add-all ms |ms. map-seq-of ms (M # Ms)}
  proof
    fix x
    assume a1: x ∈ M ** {map-add-all ms |ms. map-seq-of ms Ms}
    then obtain m ms where m1: m ∈ M and m2: map-seq-of ms Ms and m3: x = m ++ map-add-all ms
      unfolding map-Add-def by blast
    have h1: x = map-add-all (m#ms) by (simp add: add-all-left m3)
    have h2: map-seq-of (m#ms) (M # Ms)
      by (metis assms m1 m2 map-seq-in.simps(2) map-seq-of-def maps-seq-of)
    show x ∈ {map-add-all ms |ms. map-seq-of ms (M # Ms)} using h1 h2 by blast
  qed
next
  show {map-add-all ms |ms. map-seq-of ms (M # Ms)} ⊆ M ** {map-add-all ms |ms. map-seq-of ms Ms}
  proof
    fix x
    assume a1: x ∈ {map-add-all ms |ms. map-seq-of ms (M # Ms)}
   then obtain m ms where m1: m ∈ M and m2: map-seq-of (m#ms) (M # Ms) and m3: x = map-add-all
(m#ms)
      apply (simp; clarify) using map-seq-in.elims(2) map-seq-of-def by blast
    have h1: map-seq-of ms Ms using m2
      by (metis assms map-seq-in.simps(2) map-seq-of-def maps-seq-of seq-of-maps-hd)
    have h2: x = m ++ map-add-all ms by (simp add: add-all-left m3)
    show x ∈ M ** {map-add-all ms |ms. map-seq-of ms Ms}
      using h1 h2 m1 map-Add-def by fastforce
  qed
qed

lemma map-Add-seq-all: seq-of-maps Ms ⟹ map-Add-seq S Ms = S ** {map-add-all ms | ms. map-seq-of ms
Ms}
proof (induction Ms arbitrary: S)
```

```
  case Nil
  have h1: map-Add-seq S [] = S by simp
  have {map-add-all ms | ms. map-seq-of ms []} = {Map.empty}
  proof
    show {map-add-all ms |ms. map-seq-of ms []} ⊆ {Map.empty}
      using map-seq-of-empty by fastforce
  next
    show {Map.empty} ⊆ {map-add-all ms |ms. map-seq-of ms []}
      apply (simp add: map-add-all-def)
      by (metis emptyE empty-set less-nat-zero-code list.size(3) map-add-seq.simps(1)
          map-seq-in.simps(1) map-seq-of-def)
  qed
  hence S = S ** {map-add-all ms | ms. map-seq-of ms []} by (metis map-Add-unit-right)
  then show ?case using h1 by blast
next
  case (Cons a Ms)
  have h1: map-Add-seq (S ** a) Ms = (S ** a) ** {map-add-all ms | ms. map-seq-of ms Ms}
    using local.Cons seq-of-maps-hd by blast
  have h2: ... = S ** (a ** {map-add-all ms | ms. map-seq-of ms Ms}) by (simp add: map-Add-assoc)
  have h3: ... = S ** {map-add-all ms | ms. map-seq-of ms (a#Ms)}
    by (simp add: Cons.prems map-add-seq-of-shift)
  then show ?case by (simp add: h1 h2)
qed


lemma map-Add-seq-Merge:
  assumes seq-of-maps Ms
  shows map-Add-seq S Ms = S ** {⨄_m set ms | ms. map-seq-of ms Ms}
proof
  show map-Add-seq S Ms ⊆ S ** {⨄_m set ms |ms. map-seq-of ms Ms}
  proof
    fix x
    assume a1: x ∈ map-Add-seq S Ms
    hence h1: x ∈ S ** {map-add-all ms | ms. map-seq-of ms Ms} using assms map-Add-seq-all by blast
    then obtain s ms where s1: s ∈ S and s2: map-seq-of ms Ms and s3: x = s ++ map-add-all ms
      unfolding map-Add-def by blast
    have h2: map-add-all ms = ⨄_m set ms using s2 add-all[of ms] unfolding map-seq-of-def
      by (meson card-length)
    show x ∈ S ** {⨄_m set ms |ms. map-seq-of ms Ms} using h2 map-Add-def s1 s2 s3 by fastforce
  qed
next
  show S ** {⨄_m set ms |ms. map-seq-of ms Ms} ⊆ map-Add-seq S Ms
  proof
    fix x
    assume a1: x ∈ S ** {⨄_m set ms |ms. map-seq-of ms Ms}
    then show x ∈ map-Add-seq S Ms using assms add-all map-Add-seq-all[of Ms S] unfolding map-seq-of-def
      by (smt (verit) Collect-cong card-length)
  qed
qed
```

fun *map-add-seq-pair* **where**
  map-add-seq-pair (s_1, s_2) [] = (s_1, s_2)
| *map-add-seq-pair* (s_1, s_2) $((m_1, m_2)\#ms)$ = *map-add-seq-pair* $(s_1 ++ m_1, s_2 ++ m_2)$ *ms*

lemma *map-add-seq-zip*:
  ⟦*length ms* = *length ns*; *xs* = *zip ms ns* ⟧ ⟹ *map-add-seq-pair* (s_1, s_2) *xs* = (*map-add-seq* s_1 *ms*, *map-add-seq* s_2 *ns*)
proof (*induction xs arbitrary*: s_1 s_2 *ms ns*)
  case *Nil*
  then show *?case* **by** *force*
next
  case (*Cons a xs*)
  then obtain a_1 a_2 **where** *p1*: *a* = (a_1, a_2) **using** *old.prod.exhaust* **by** *blast*
  obtain *ms'* **where** *p2*: *ms* = $a_1\#ms'$ **by** (*metis Cons.prems*(2) *p1 prod.inject zip-eq-ConsE*)
  obtain *ns'* **where** *p3*: *ns* = $a_2\#ns'$ **by** (*metis Cons.prems*(2) *p1 prod.inject zip-eq-ConsE*)
  have *map-add-seq-pair* (s_1, s_2) $(a\#xs)$ = *map-add-seq-pair* $(s_1 ++ a_1, s_2 ++ a_2)$ *xs* **by** (*simp add*: *p1*)
  then show *?case* **using** *Cons.IH Cons.prems*(1) *Cons.prems*(2) *p2 p3* **by** *force*
qed

lemma *map-add-seq-Merge-pair*:
  assumes *dM*: ∀ $m_1 \in M$.∀ $m_2 \in M$. *dom* m_1 ∩ *dom* m_2 = {}
    and *dN*: ∀ $n_1 \in N$.∀ $n_2 \in N$. *dom* n_1 ∩ *dom* n_2 = {}
    and *sM*: *M* = *set ms*
    and *sN*: *N* = *set ns*
    and *cM*: *card M* = *length ms*
    and *cN*: *card N* = *length ns*
    and *cC*: *card M* = *card N*
   shows *map-add-seq-pair* (sm, sn) (*zip ms ns*) = $(sm ++ (\biguplus_m M), sn ++ (\biguplus_m N))$
  using *assms* **by** (*metis map-add-seq-Merge map-add-seq-zip nth-mem*)

lemma *map-update-merge*:
  assumes *d*: *a* ∉ *dom m*
  shows $[a\mapsto b] ++ m = [a\mapsto b] \uplus_m m$
  by (*simp add*: *d map-add-merge*)

fun *map-upd-seq* **where**
  map-upd-seq f s [] = *s*
| *map-upd-seq f s* (*m*#*ms*) = *map-upd-seq f* $(s(m\mapsto f\ m))$ *ms*

lemma *map-upd-seq-foldl*: *map-upd-seq f s ms* = *foldl* $(\lambda s\ a.\ s(a\mapsto f\ a))$ *s ms*
proof (*induction ms arbitrary*: *s*)
  case *Nil*
  then show *?case* **by** *fastforce*
next
  case (*Cons a ms*)
  then show *?case* **by** *simp*
qed

lemma *map-upd-seq-add*: *map-upd-seq f s ms* = *map-add-seq s* (*map* $(\lambda a.\ [a\mapsto f\ a])$ *ms*)
proof (*induction ms arbitrary*: *s*)

```
  case Nil
  then show ?case by simp
next
  case (Cons a ms)
  have map-upd-seq f (s(a↦f a)) ms = map-add-seq (s(a↦f a)) (map (λa. [a ↦ f a]) ms)
    using local.Cons by blast
  then show ?case unfolding map-upd-seq-foldl
    by (smt (verit, best) foldl-Cons list.distinct(1) list.inject list.simps(9)
        map-add-seq.elims singleton-map-upd)
qed

lemma map-upd-Merge:
  assumes c: card (set xs) = length xs
    shows map-upd-seq f s xs = s ++ (⨄ m { [x ↦ f x] | x. x ∈ set xs })
proof −
  have h0: map-upd-seq f s xs = map-add-seq s (map (λa. [a ↦ f a]) xs)
  proof (induction xs arbitrary: s)
    case Nil
    then show ?case by simp
  next
    case (Cons a xs)
    then show ?case using map-upd-seq-add by blast
  qed
  have h4: set (map (λa. [a ↦ f a]) xs) = { [x ↦ f x] | x. x ∈ set xs }
  proof (induction xs)
    case Nil
    then show ?case by force
  next
    case (Cons a xs)
    have x1: set (map (λa. [a ↦ f a]) (a#xs)) ⊆ { [x ↦ f x] | x. x ∈ set (a#xs) }
    proof
      fix x
      assume a1: x ∈ set (map (λa. [a ↦ f a]) (a # xs))
      show x ∈ {[x ↦ f x] |x. x ∈ set (a # xs)} using a1 by auto
    qed
    have x2: { [x ↦ f x] | x. x ∈ set (a#xs) } ⊆ set (map (λa. [a ↦ f a]) (a#xs))
    proof
      fix x
      assume a1: x ∈ {[x ↦ f x] |x. x ∈ set (a # xs)}
      show x ∈ set (map (λa. [a ↦ f a]) (a # xs)) using a1 by auto
    qed
    show ?case using x1 x2 by blast
  qed
  obtain ms where ms1: ms = map (λa. [a ↦ f a]) xs by blast
  have h1: ∀ i j. i < length ms ∧ j < length ms ∧ dom (ms!i) ∩ dom (ms!j) ≠ {} ⟶ i = j
    using c apply (simp add: ms1 singleton-unfold; clarify)
    by (smt (verit, best) card-distinct disjoint-iff domIff nth-eq-iff-index-eq nth-map)
  hence h2: map-add-seq s ms = s ++ (⨄ m set ms) by (simp add: map-add-seq-Merge)
  hence h3: ... = s ++ (⨄ m { [x ↦ f x] | x. x ∈ set xs }) using h4 ms1 by presburger
  show ?thesis using h0 h2 h4 ms1 by auto
```

qed

fun *map-Upd-seq* **where**
  map-Upd-seq f S [] = *S*
| *map-Upd-seq f S* (*m*#*ms*) = *map-Upd-seq f* { *s*(*m*↦*x*) |*s x*. *s* ∈ *S* ∧ *x* ∈ *f m* } *ms*

definition *maps-of* **where** *maps-of f x* = { [*x* ↦ *y*] | *y*. *y* ∈ *f x* }

lemma *map-maps-hd*: *map* (*maps-of f*) (*x*#*xs*) = { [*x* ↦ *y*] | *y*. *y* ∈ *f x* }#*map* (*maps-of f*) *xs*
  by (*simp add*: *maps-of-def*)

lemma *Dom-maps-of*:*Dom* (⋃ (*set* (*map* (*maps-of f*) *xs*))) ⊆ *set xs*
proof (*induction xs*)
  case *Nil*
  then show *?case* **unfolding** *Dom-def dom-def maps-of-def* **by** (*simp*; *clarify*)
next
  case (*Cons a xs*)
  then show *?case* **unfolding** *Dom-def dom-def maps-of-def* **apply** *simp* **by** *fastforce*
qed

lemma *Dom-maps-of-inner*:∀ *x* ∈ *set xs*. *f x* ≠ {} ⟹ *Dom* (⋃ (*set* (*map* (*maps-of f*) *xs*))) = *set xs*
proof (*induction xs*)
  case *Nil*
  then show *?case* **using** *Dom-def* **by** *fastforce*
next
  case (*Cons a xs*)
  have *h1*: *Dom* (⋃ (*set* (*map* (*maps-of f*) *xs*))) = *set xs* **using** *Cons.IH Cons.prems* **by** *force*
  have *h2*: *Dom* (⋃ (*set* (*map* (*maps-of f*) (*a* # *xs*)))) =
      Dom (⋃ (*set* ({ [*a* ↦ *y*] | *y*. *y* ∈ *f a* }#*map* (*maps-of f*) *xs*)))
    by (*metis map-maps-hd*)
  have *h3*: ... = *Dom* (⋃ ({{ [*a* ↦ *y*] | *y*. *y* ∈ *f a* }} ∪ *set* (*map* (*maps-of f*) *xs*))) **by** *force*
  have *h4*: ... = *Dom* ({ [*a* ↦ *y*] | *y*. *y* ∈ *f a* } ∪ ⋃ (*set* (*map* (*maps-of f*) *xs*))) **by** *force*
  have *h5*: ... = {*a*} ∪ *Dom* (⋃ (*set* (*map* (*maps-of f*) *xs*)))
    unfolding *Dom-def dom-def* **apply** *simp* **apply** (*rule antisym*)
    using *SUP-le-iff* **apply** *fastforce*
    apply (*simp*; *clarify*)
    by (*metis Cons.prems ex-in-conv fun-upd-same list.set-intros*(*1*))
  then show *?case* **using** *h1 h2* **by** *force*
qed

lemma *Dom-maps-of-outer*:⟦ ∀ *x* ∈ *set xs*. *f x* ≠ {}; *k* < *length xs* ⟧ ⟹ *Dom* ((*map* (*maps-of f*) *xs*)!*k*) ⊆ *set xs*
proof (*induction xs arbitrary*: *k*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
    then show *?case*
    proof (*cases k* = *0*)
      case *True*
      have *h1*: *Dom* (*map* (*maps-of f*) (*a* # *xs*) ! *k*) = *Dom* (({ [*a* ↦ *y*] | *y*. *y* ∈ *f a* }#*map* (*maps-of f*) *xs*) ! *k*)

```
      by (metis map-maps-hd)
    have h2: ... = Dom ({ [a ↦ y] | y. y ∈ f a }) by (simp add: True)
    have h3: ... = {a} using Cons.prems(1) by (simp add: Dom-single ex-in-conv)
    then show ?thesis using h1 h2 by force
  next
    case False
    have h1: Dom (map (maps-of f) (a # xs) ! k) = Dom (({ [a ↦ y] | y. y ∈ f a }#map (maps-of f) xs) ! k)
      by (metis map-maps-hd)
    have h2: ... = Dom ((map (maps-of f) xs) ! (k−1))
      by (simp add: False zero-less-iff-neq-zero)
    then show ?thesis
      using Cons.IH Cons.prems(1) Cons.prems(2) False apply (simp; clarify)
      by (metis Suc-less-eq Suc-pred subsetD)
  qed
qed

lemma Dom-maps-of-diff:⟦ card (set xs) = length xs; ∀ x ∈ set xs. f x ≠ {}; i < length xs; j < length xs;
  i ≠ j ⟧ ⟹ Dom ((map (maps-of f) xs)!i) ∩ Dom ((map (maps-of f) xs)!j) = {}
proof (induction xs arbitrary: i j)
  case Nil
  then show ?case
    by simp
next
  case (Cons a xs)
  then show ?case
  proof (cases i = 0)
    case True
    hence h1: Dom (map (maps-of f) (a # xs) ! i) = {a}
      unfolding Dom-def dom-def maps-of-def apply simp apply (rule antisym)
      apply clarify
       apply (metis option.distinct(1) singleton-unfold)
      apply clarify using Cons.prems(2) by auto
    have h2: j > 0 using Cons.prems(5) True by force
    hence h3: Dom (map (maps-of f) (a # xs) ! j) = Dom (({ [a ↦ y] | y. y ∈ f a }#map (maps-of f) xs) ! j)
      by simp
    have h4: ... = Dom (map (maps-of f) xs ! (j−1)) using h2 by force
    have h5: ... ⊆ set xs
      using h2
      by (metis Cons.prems(2) Cons.prems(4) Dom-maps-of-outer One-nat-def Suc-less-eq Suc-pred
          length-Cons set-subset-Cons subsetD)
    then show ?thesis
      by (metis Cons.prems(1) Int-insert-left-if0 card-distinct distinct.simps(2) h1 h3 h4
          inf-bot-left subsetD)
  next
    case False
    hence h0: i > 0 by simp
    then show ?thesis
    proof (cases j = 0)
      case True
    hence h1: Dom (map (maps-of f) (a # xs) ! j) = {a}
```

```
      unfolding Dom-def dom-def maps-of-def apply simp apply (rule antisym)
      apply clarify
       apply (metis option.distinct(1) singleton-unfold)
      apply clarify using Cons.prems(2) by auto
    have h2: Dom (map (maps-of f) (a # xs) ! i) = Dom (({ [a ↦ y] | y. y ∈ f a }#map (maps-of f) xs) ! i)
      by (metis map-maps-hd)
    have h4: ... = Dom (map (maps-of f) xs ! (i−1)) using h1 False by force
    have h5: ... ⊆ set xs
      by (metis Cons.prems(2) Cons.prems(3) Dom-maps-of-outer False One-nat-def Suc-less-eq
          Suc-pred bot-nat-0.not-eq-extremum length-Cons set-subset-Cons subset-iff)
      then show ?thesis
        by (metis Cons.prems(1) Int-insert-right card-distinct distinct.simps(2) h1 h2 h4 inf-bot-right subsetD)
    next
      case False
      have h6: Dom (map (maps-of f) (a # xs) ! i) = Dom (({ [a ↦ y] | y. y ∈ f a }#map (maps-of f) xs) ! i)
        by (simp add: maps-of-def)
      have i6: ... = Dom (map (maps-of f) xs ! (i−1)) by (simp add: h0)
      have h7: Dom (map (maps-of f) (a # xs) ! j) = Dom (({ [a ↦ y] | y. y ∈ f a }#map (maps-of f) xs) ! j)
        by (simp add: maps-of-def)
      have i7: ... = Dom (map (maps-of f) xs ! (j−1)) by (simp add: False zero-less-iff-neq-zero)
      have x1: i−1 < length xs using Cons.prems(3) h0 by auto
      have x2: j−1 < length xs using Cons.prems(4) False by force
      have x3: i−1 ≠ j−1 by (metis Cons.prems(5) False Suc-pred' gr-zeroI h0)
      have x4: Dom (map (maps-of f) xs ! (i−1)) ∩ Dom (map (maps-of f) xs ! (j−1)) = {}
        by (metis Cons.IH Cons.prems(1) Cons.prems(2) card-distinct distinct.simps(2) distinct-card
            list.set-intros(2) x1 x2 x3)
      then show ?thesis using h6 h7 i6 i7 by blast
    qed
  qed
qed

lemma map-Upd-seq-comp: map-Upd-seq f (map-Upd-seq f S xs) ys = map-Upd-seq f S (xs @ ys)
proof (induction xs arbitrary: S)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  then show ?case by simp
qed

lemma map-Upd-comp-mono: S ⊆ T ⟹ map-Upd-seq f S xs ⊆ map-Upd-seq f T xs
proof (induction xs arbitrary: S T)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  show ?case
  proof
    fix x
    assume x ∈ map-Upd-seq f S (a # xs)
```

```
    hence x ∈ map-Upd-seq f {s(a ↦ x) |s x. s ∈ S ∧ x ∈ f a} xs
      by simp
    hence x ∈ map-Upd-seq f {s(a ↦ x) |s x. s ∈ T ∧ x ∈ f a} xs
      using Cons.IH[of {s(a ↦ x) |s x. s ∈ S ∧ x ∈ f a} {s(a ↦ x) |s x. s ∈ T ∧ x ∈ f a}] Cons.prems by blast
    then show x ∈ map-Upd-seq f T (a # xs) using Cons.prems Cons.IH by simp
  qed
qed


lemma map-Upd-seq-comp-in: ⟦x ∈ (map-Upd-seq f S xs); y ∈ map-Upd-seq f {x} ys⟧ ⟹ y ∈ map-Upd-seq f S
(xs @ ys)
proof (induction xs arbitrary: S)
  case Nil
  then show ?case
   by (metis (no-types, opaque-lifting) empty-subsetI in-mono insert-subsetI map-Upd-comp-mono map-Upd-seq-comp)
next
  case (Cons a xs)
  then show ?case by simp
qed

lemma map-Upd-one: map-Upd-seq f {a} [m] = { a(m↦x) |x. x ∈ f m } by force

lemma map-Upd-Add:
  assumes c: card (set xs) = length xs
  shows map-Upd-seq f S xs = map-Add-seq S (map (maps-of f) xs)
proof (induction xs arbitrary: S)
  case Nil
  then show ?case by simp
next
  case (Cons a xs)
  have h1: map-Upd-seq f (S ** { [a ↦ y] | y. y ∈ f a }) xs =
        map-Add-seq (S ** { [a ↦ y] | y. y ∈ f a }) (map (maps-of f) xs) using local.Cons by blast
  have h2: map-Upd-seq f S (a # xs) = map-Upd-seq f { s(a↦y) |s y. s ∈ S ∧ y ∈ f a } xs by simp
  have h3: { s(a↦y) |s y. s ∈ S ∧ y ∈ f a } = S ** { [a ↦ y] | y. y ∈ f a }
  proof
    show {s(a ↦ y) |s y. s ∈ S ∧ y ∈ f a} ⊆ S ** {[a ↦ y] |y. y ∈ f a}
      apply (simp add: map-Add-def; clarify) by force
  next
    show S ** {[a ↦ y] |y. y ∈ f a} ⊆ {s(a ↦ y) |s y. s ∈ S ∧ y ∈ f a}
      apply (simp add: map-Add-def; clarify) by auto
  qed
  have h4: map-Add-seq (S ** { [a ↦ y] | y. y ∈ f a }) (map (maps-of f) xs) =
        map-Add-seq S (map (maps-of f) (a # xs))
    by (simp add: maps-of-def)
  then show ?case using h1 h2 h3 by presburger
qed

lemma ⟦ k < length xs; x ∉ set xs ⟧ ⟹ x ∉ Dom (map (maps-of f) xs ! k)
proof (induction xs arbitrary: k)
  case Nil
```

```
  then show ?case
    unfolding Dom-def dom-def by fastforce
next
  case (Cons a xs)
  have k1: k = 0 ⟹ Dom (map (maps-of f) (a # xs) ! k) ⊆ {a}
    apply (simp add: map-maps-hd[of f a xs] maps-of-def Dom-def dom-def; clarify)
    by (metis domI domIff singleton-unfold)
  have k2: k ≠ 0 ⟹ Dom (map (maps-of f) (a # xs) ! k) ⊆ Dom (map (maps-of f) xs ! (k−1)) by auto
  show ?case using Cons.IH Cons.prems(1) Cons.prems(2) k1 k2
    apply (simp; clarify)
    using less-Suc-eq-0-disj by auto
qed

lemma seq-of-map-maps:
  ⟦card (set xs) = length xs; ∀ x ∈ set xs. f x ≠ {}⟧ ⟹ seq-of-maps (map (maps-of f) xs)
proof (induction xs)
  case Nil
  then show ?case by (simp add: seq-of-maps-def)
next
  case (Cons a xs)
  have h1: seq-of-maps (map (maps-of f) xs)
    by (meson Cons.IH Cons.prems(1) Cons.prems(2) card-distinct distinct.simps(2) distinct-card
        list.set-intros(2))
  have h2: ∀ i j. i < length (map (maps-of f) (a # xs)) ∧ j < length (map (maps-of f) (a # xs)) ∧
      Dom ((map (maps-of f) (a # xs))!i) ∩ Dom ((map (maps-of f) (a # xs))!j) ≠ {} ⟶ i = j
    using Dom-maps-of-diff by (metis Cons.prems(1) Cons.prems(2) length-map)
  have h3: ∀ M ∈ set (map (maps-of f) (a # xs)).∀ m₁ ∈ M.∀ m₂ ∈ M. dom m₁ = dom m₂
    using h1 unfolding maps-of-def seq-of-maps-def apply clarify by fastforce
  then show ?case by (metis h2 seq-of-maps-def)
qed

lemma map-Upd-Merge:
  assumes c: card (set xs) = length xs
  assumes f: ∀ x ∈ set xs. f x ≠ {}
  shows map-Upd-seq f S xs = S ** {⨄ₘ set ms |ms. map-seq-in ms (map (maps-of f) xs)}
proof
  show map-Upd-seq f S xs ⊆ S ** {⨄ₘ set ms |ms. map-seq-in ms (map (maps-of f) xs)}
  proof
    fix x
    assume a1: x ∈ map-Upd-seq f S xs
    hence h1: x ∈ map-Add-seq S (map (maps-of f) xs) using c map-Upd-Add by blast
    have h2: seq-of-maps (map (maps-of f) xs) using c f seq-of-map-maps by blast
    hence h2: x ∈ S ** {⨄ₘ set ms | ms. map-seq-of ms (map (maps-of f) xs)}
      using h1 map-Add-seq-Merge by blast
    show x ∈ S ** {⨄ₘ set ms |ms. map-seq-in ms (map (maps-of f) xs)}
      using c f h2 map-seq-of-wk[of - map (maps-of f) xs]
        maps-seq-of[of map (maps-of f) xs] seq-of-map-maps[of xs f]
      by (smt (verit, best) Collect-cong)
  qed
next
```

```
  show S ** {⨄_m set ms |ms. map-seq-in ms (map (maps-of f) xs)} ⊆ map-Upd-seq f S xs
    using c f map-Upd-Add[of xs f S] map-seq-of-wk[of - map (maps-of f) xs]
      maps-seq-of[of map (maps-of f) xs] seq-of-map-maps[of xs f]
    map-Add-seq-Merge[of map (maps-of f) xs S] apply clarify
    by (smt (z3) CollectD CollectI map-Add-def)
qed

lemma map-seq-in-dom: ⟦map-seq-in ms (map (maps-of f) xs); m ∈ set ms⟧ ⟹ dom m ⊆ set xs
proof (induction xs arbitrary: ms)
  case Nil
  then show ?case using map-seq-in-length by fastforce
next
  case (Cons a xs)
  obtain z zs where z1: ms = z#zs by (meson Cons.prems(2) list.set-cases)
  hence h1: m ∈ set zs ⟹ dom m ⊆ set xs using Cons.prems Cons.IH apply simp by blast
  have h2: map (maps-of f) (a # xs) = {[a ↦ y] |y. y ∈ f a}#map (maps-of f) xs
    by (metis map-maps-hd)
  have dom z ⊆ {a} using Cons.prems(1) apply (simp add: h2 z1 maps-of-def)
    using dom-eq-singleton-conv by force
  then show ?case using Cons.prems(2) h1 z1 by auto
qed

lemma map-seq-merge-el: ⟦map-seq-in ms (map (maps-of f) xs); c ∈ set xs; card (set xs) = length xs⟧ ⟹ (⨄_m
set ms) $ c ∈ f c
proof (induction xs arbitrary: ms)
  case Nil
  then show ?case using map-seq-in-length by fastforce
next
  case (Cons a as)
  obtain z zs where z1: ms = z#zs
    by (metis list.discI list.simps(9) local.Cons(2) map-seq-in.elims(2))
  hence z2: map-seq-in zs (map (maps-of f) as) using Cons.prems(1) by fastforce
  have h2: map (maps-of f) (a # xs) = {[a ↦ y] |y. y ∈ f a}#map (maps-of f) xs
    by (metis map-maps-hd)
  have z3: z $ a ∈ f a
    using Cons.prems apply (simp add: z1)
    unfolding maps-of-def apply clarify
    by simp
  have h3: z ∈ {[a ↦ y] |y. y ∈ f a} using Cons.prems(1) h2 z1 by force
  have z4: card (set as) = length as
    by (meson Cons.prems(3) card-distinct distinct.simps(2) distinct-card)
  have z5: c ∈ set as ⟹ ⨄_m set zs $ c ∈ f c using Cons.IH z2 z4 by blast
  have z6: ∀ x ∈ set zs. dom z ∩ dom x = {}
    using h3 z2 apply simp
    by (metis Cons.prems(3) Int-emptyI card-distinct distinct.simps(2) dom-empty dom-fun-upd
        map-seq-in-dom option.distinct(1) singletonD subsetD)
  have z7: dom (⨄_m set zs) ∩ dom z = {}
    by (metis inf.idem inf-bot-right inf-commute not-dom-Merge z6)
  hence z8: (⨄_m set zs ⊎_m z) $ c ∈ f c
  proof (cases c ∈ set as)
```

```
    case True
    then show ?thesis using h3 z5 z6 z7
      by (smt (verit) CollectD Cons.prems(3) card-distinct distinct.simps(2) domIff
          map-add-dom-app-simps(3) map-add-merge map-get-def singleton-unfold)
  next
    case False
    then show ?thesis using h3 z3 z6 z7
      by (smt (verit, best) Cons.prems(2) domI fun-upd-same inf.commute map-add-comm map-add-merge
          map-some-val mem-Collect-eq set-ConsD map-le-def map-merge-left-sub)
  qed
  then show ?case
    by (smt (verit, best) Diff-insert-absorb dom-eq-empty-conv inf.idem list.set-intros(1)
        list.simps(15) merge-Merge-diff merge-unit remdups.simps(2) set-remdups z1 z6)
qed

lemma map-seq-fun-dep:
  assumes card (set xs) = length xs
      and c ∈ set xs
      and m ∈ {⨄ m set ms |ms. map-seq-in ms (map (maps-of f) xs)}
    shows m $ c ∈ f c
  using assms map-seq-merge-el by fastforce

lemma map-seq-merge-subset: ⟦map-seq-in ms (map (maps-of f) xs)⟧ ⟹ dom (⨄ m set ms) ⊆ set xs
proof (induction xs arbitrary: ms)
  case Nil
  then show ?case using map-seq-in-length by fastforce
next
  case (Cons a xs)
  then show ?case by (smt (verit) UN-least dual-order.trans map-Merge-dom-sub map-seq-in-dom)
qed

lemma map-seq-dom-sub:
  assumes m ∈ {⨄ m set ms |ms. map-seq-in ms (map (maps-of f) xs)}
    shows dom m ⊆ set xs
  using assms map-seq-merge-subset by blast

lemma map-seq-merge-eq: ⟦map-seq-in ms (map (maps-of f) xs); card (set xs) = length xs⟧ ⟹ dom (⨄ m set
ms) = set xs
proof
  show ⟦map-seq-in ms (map (maps-of f) xs); card (set xs) = length xs ⟧ ⟹ dom (⨄ m set ms) ⊆ set xs
    by (simp add: map-seq-merge-subset)
next
  show ⟦map-seq-in ms (map (maps-of f) xs); card (set xs) = length xs⟧ ⟹ set xs ⊆ dom (⨄ m set ms)
  proof (induction xs arbitrary: ms)
    case Nil
    then show ?case by simp
  next
    case (Cons a as)
    obtain z zs where z1: ms = z#zs
      by (meson Cons.prems list.discI list.map-disc-iff map-seq-in.elims(2))
```

```
    have z2: map (maps-of f) (a # as) = {[a ↦ y] |y. y ∈ f a}#map (maps-of f) as
      by (metis map-maps-hd)
    have z3: map-seq-in zs (map (maps-of f) as) using Cons.prems z1 by auto
    then have z4: set as ⊆ dom (⨄m set zs) using Cons.IH
      by (meson Cons.prems(2) card-distinct distinct.simps(2) distinct-card)
    have z5: z ∈ {[a ↦ y] |y. y ∈ f a} using Cons.prems z1 z2 by force
    then have z6: a ∈ dom z by fastforce
    have z7: ∀ x ∈ set zs. dom z ∩ dom x = {} using Cons.prems(2) z3 z5 apply simp
      by (metis Cons.prems(2) IntD2 card-distinct distinct.simps(2) dom-eq-singleton-conv
          empty-subsetI insert-disjoint(2) le-iff-inf map-seq-in-dom)
    have a1: a ∉ set as by (meson Cons.prems(2) card-distinct distinct.simps(2))
    have a2: a ∉ dom (⨄m set zs) using a1 map-seq-merge-subset z3 by blast
    have a3: ⨄m set ms = ⨄m (set zs ∪ {z}) by (simp add: z1)
    have a4: ... = ⨄m set zs ⊎m z using merge-Merge-union z7 by blast
    show ?case using a3 a4 z3 z4 z6 z7
      by (smt (verit, ccfv-threshold) inf-commute insert-Diff insert-disjoint(2) insert-subset
          list.simps(15) map-le-implies-dom-le map-merge-left-sub map-merge-right-sub
          map-seq-merge-subset not-dom-Merge subset-antisym)
  qed
qed

lemma map-seq-dom-dep:
  assumes card (set xs) = length xs
      and m ∈ {⨄m set ms |ms. map-seq-in ms (map (maps-of f) xs)}
    shows dom m = set xs
  using assms map-seq-merge-eq by fastforce

end
```

Part IV

Examples

Chapter 8

Model Examples

This chapter includes examples of how the HAMR model-driven development tool chain translates AADL instance model information into the types defined in Model.thy. Along with the model information, lemmas for model well-formedness are generated, and these are automatically proved by Isabelle. This information is derived directly from and is traceable to the representation of AADL model information in HAMR-generated code.

8.1 Temperature Control Example

8.1.1 AADL Model Overview

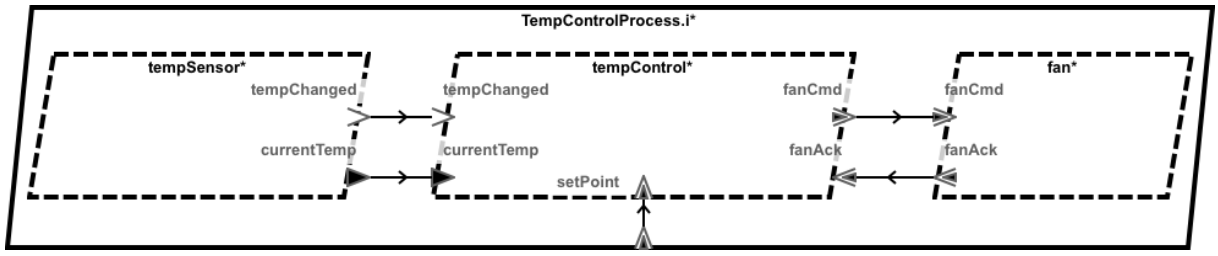

Figure 8.1: Temperature Control Example – AADL Graphical View

Figure 8.1 presents the AADL graphical view for a simple temperature control system that maintains a temperature according to a set point containing high and low bounds for the target temperature. The periodic `tempSensor` thread measures the temperature and transmits the reading on its `currentTemp` data port. It sends a notification on its `tempChanged` event port if it detects that the temperature has changed since the last reading. When the sporadic (event-driven) `tempControl` thread receives a `tempChanged` event, it reads the value on its `currentTemp` data port and compares it with the most recent set point. If the current temperature exceeds the high set point or drops below the low set point, the fan is turned on or off respectively. In turn, the fan acknowledges these commands.

```
thread TempControl
  features
  -- ==== INPUTS ====
    currentTemp: in data port TempSensor::Temperature.i;
    tempChanged: in event port;
    fanAck: in event data port CoolingFan::FanAck;
    setPoint: in event data port SetPoint.i;
    -- ==== OUTPUTS ====
    fanCmd: out event data port CoolingFan::FanCmd;
  properties
    Dispatch_Protocol => Sporadic;
    Period => 1 sec;
end TempControl
```

Figure 8.2: Temperature Control Thread – AADL Textual View

AADL's textual view for the component type definition of the `TempControl` thread is shown in Figure 8.2. Because `TempControl` is event-triggered, it will be dispatched by arrival of events on any of the `tempChanged`, `fanAck`, `setPoint` ports. Most interesting for us is the dispatching on the arrival of the `tempChanged` event. In this case, the thread application logic will read the `currentTemp` data port, compare the value to the most recent `setPoint` values, and compute an appropriate state for the fan, sending an on/off command over the `fanCmd` port if necessary.

Isabelle Model Representation

theory *TempControlModel*
imports *Model*
begin

The following are *PortDescr* definitions for the ports of the `TempSensor` component.

PortDescr for the `currrentTemp` port. The names and ids for ports and components are generated from what HAMR uses for code generation.

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-currentTemp* **where**
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-currentTemp =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-currentTemp'' — Fully-qualified port name.
(*PortId 0*) — Port identifier for this port.
(*CompId 0*) — Component identifier for the parent component for this port.
Out — Port direction.
Data — Port kind.
1 — Maximum number of values in port buffer.
0 — Urgency (priority).

Add this definition to the list of definitions that will be automatically unfolded by the Isabelle simplifier. This is primarily used to automatically prove model well-formedness using the "simp" tactic.

declare *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-currentTemp-def* [*simp add*]

PortDescr for the `tempChanged` port.

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-tempChanged* **where**

TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-tempChanged =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-tempChanged'' — Fully-qualified port name.
(*PortId 1*) — Port identifier for this port.
(*CompId 0*) — Component identifier for the parent component for this port.
Out — Port direction.
Event — Port kind.
1 — Maximum number of values in port buffer.
0 — Urgency (priority).
declare *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-tempChanged-def* [*simp add*]

Now, we have the *CompDescr* definition for the `TempSensor` component.

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor* **where**
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor =
mkCompDescr
''TempControlSoftwareSystem-s-Instance-tcproc-tempSensor'' — Fully-qualified component name.
(*CompId 0*) — Component identifier for this component.
{(*PortId 0*),(*PortId 1*)} — Set of identifiers of ports that belong to this component.
DispatchProtocol.Periodic — Dispatch protocol.
{} — Identifiers of ports that are dispatch triggers.
{} — Identifiers of component local variables.
declare *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-def* [*simp add*]

Explanations for other ports and components are similar.

definition *TempControlSoftwareSystem-s-Instance-tcproc-fan-fanCmd* **where**
TempControlSoftwareSystem-s-Instance-tcproc-fan-fanCmd =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-fan-fanCmd''
(*PortId 2*)
(*CompId 1*)
In
Event
1
0
declare *TempControlSoftwareSystem-s-Instance-tcproc-fan-fanCmd-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-fan-fanAck* **where**
TempControlSoftwareSystem-s-Instance-tcproc-fan-fanAck =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-fan-fanAck''
(*PortId 3*)
(*CompId 1*)
Out
Event
1
0
declare *TempControlSoftwareSystem-s-Instance-tcproc-fan-fanAck-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-fan* **where**
TempControlSoftwareSystem-s-Instance-tcproc-fan =
mkCompDescr
''TempControlSoftwareSystem-s-Instance-tcproc-fan''
(*CompId 1*)
{(*PortId 2*),(*PortId 3*)}
DispatchProtocol.Sporadic
{(*PortId 2*)}
{}
declare *TempControlSoftwareSystem-s-Instance-tcproc-fan-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-currentTemp* **where**
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-currentTemp =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-tempControl-currentTemp''
(*PortId 4*)
(*CompId 2*)
In
Data
1
0
declare *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-currentTemp-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-fanAck* **where**
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-fanAck =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-tempControl-fanAck''
(*PortId 5*)
(*CompId 2*)
In
Event
1
0
declare *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-fanAck-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-setPoint* **where**
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-setPoint =
mkPortDescr

''TempControlSoftwareSystem-s-Instance-tcproc-tempControl-setPoint''
(PortId 6)
(CompId 2)
In
Event
1
0
declare *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-setPoint-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-tempChanged* **where**
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-tempChanged =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-tempControl-tempChanged''
(PortId 8)
(CompId 2)
In
Event
1
0
declare *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-tempChanged-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-fanCmd* **where**
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-fanCmd =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-tempControl-fanCmd''
(PortId 7)
(CompId 2)
Out
Event
1
0
declare *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-fanCmd-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl* **where**
TempControlSoftwareSystem-s-Instance-tcproc-tempControl =
mkCompDescr
''TempControlSoftwareSystem-s-Instance-tcproc-tempControl''
(CompId 2)
{(PortId 4),(PortId 5),(PortId 6),(PortId 8),(PortId 7)}
DispatchProtocol.Sporadic
{(PortId 5),(PortId 6),(PortId 8)}
{}
declare *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-currentTemp* **where**
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-currentTemp =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-currentTemp''
(*PortId 9*)
(*CompId 3*)
In
Data
1
0
declare *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-currentTemp-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-tempChanged* **where**
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-tempChanged =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-tempChanged''
(*PortId 11*)
(*CompId 3*)
In
Event
1
0
declare *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-tempChanged-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-setPoint* **where**
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-setPoint =
mkPortDescr
''TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-setPoint''
(*PortId 10*)
(*CompId 3*)
Out
Event
1
0
declare *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-setPoint-def* [*simp add*]

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface* **where**
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface =
mkCompDescr

''TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface''
(*CompId 3*)
{(*PortId 9*),(*PortId 11*),(*PortId 10*)}
DispatchProtocol.Periodic
{}
{}
declare *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-def* [*simp add*]

The definition below specifies the connections of the system.

definition *sysConns*
where *sysConns = map-of* [
((*PortId 0*), {(*PortId 4*),(*PortId 9*)}), — TempSensor currentTemp is connected to currentTemp ports for TempControl and OperatorInterface.
((*PortId 1*), {(*PortId 8*),(*PortId 11*)}), — TempSensor tempChanged is connected to tempChanged ports for TempControl and OperatorInterface.
((*PortId 3*), {(*PortId 5*)}), — TempControl fanCmd is connected to fanCmd port of Fan component.
((*PortId 7*), {(*PortId 2*)}), — Fan ack is connected to ack port of TempControl component.
((*PortId 10*), {(*PortId 6*)}) — OperatorInterface setPoint is connected to setPoint port of TempControl component.
]
declare *sysConns-def* [*simp add*]

The definition below maps each port identifier to a port descriptor.

definition *sysPortDescrs*
where *sysPortDescrs = map-of* [
((*PortId 0*), *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-currentTemp*),
((*PortId 1*), *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-tempChanged*),
((*PortId 2*), *TempControlSoftwareSystem-s-Instance-tcproc-fan-fanCmd*),
((*PortId 3*), *TempControlSoftwareSystem-s-Instance-tcproc-fan-fanAck*),
((*PortId 4*), *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-currentTemp*),
((*PortId 5*), *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-fanAck*),
((*PortId 6*), *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-setPoint*),
((*PortId 8*), *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-tempChanged*),
((*PortId 7*), *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-fanCmd*),
((*PortId 9*), *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-currentTemp*),
((*PortId 11*), *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-tempChanged*),
((*PortId 10*), *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-setPoint*)
]
declare *sysPortDescrs-def* [*simp add*]

The definition below maps each component identifier to a component descriptor.

definition *sysCompDescrs*
where *sysCompDescrs = map-of* [
((*CompId 0*), *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor*),
((*CompId 1*), *TempControlSoftwareSystem-s-Instance-tcproc-fan*),
((*CompId 2*), *TempControlSoftwareSystem-s-Instance-tcproc-tempControl*),
((*CompId 3*), *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface*)
]
declare *sysCompDescrs-def* [*simp add*]

The definition below is the top-level model structure for the system.

definition *sysModel*
  where *sysModel = mkModel sysCompDescrs sysPortDescrs sysConns*
declare *sysModel-def* [*simp add*]

The following definitions establish various model well-formedness properties.

lemma *sysModel-wf-Model-PortDescr*: *wf-Model-PortDescr sysModel*
  by (*simp add*: *wf-Model-PortDescr-def wf-PortDescr-def*)

lemma *sysModel-wf-Model-PortDescrsIds*: *wf-Model-PortDescrsIds sysModel*
  by (*simp add*: *wf-Model-PortDescrsIds-def*)

lemma *sysModel-wf-Model-CompDescrsIds*: *wf-Model-CompDescrsIds sysModel*
  by (*simp add*: *wf-Model-CompDescrsIds-def*)

lemma *sysModel-wf-Model-PortDescrsCompId*: *wf-Model-PortDescrsCompId sysModel*
  by (*simp add*: *wf-Model-PortDescrsCompId-def*)

lemma *sysModel-wf-Model-CompDescrsContainedPortIds*: *wf-Model-CompDescrsContainedPortIds sysModel*
  by (*simp add*: *wf-Model-CompDescrsContainedPortIds-def*)

lemma *sysModel-wf-Model-ConnsPortIds*: *wf-Model-ConnsPortIds sysModel*
  by (*simp add*: *wf-Model-ConnsPortIds-def*)

lemma *sysModel-wf-Model-DisjointPortIds*: *wf-Model-DisjointPortIds sysModel*
  by (*simp add*: *wf-Model-DisjointPortIds-def*)

lemma *sysModel-wf-Model-ConnsPortCategories*: *wf-Model-ConnsPortCategories sysModel*
  by (*simp add*: *wf-Model-ConnsPortCategories-def*)

lemma *sysModel-wf-Model-InDataPorts*: *wf-Model-InDataPorts sysModel*
  by (*simp add*: *wf-Model-InDataPorts-def*)

lemma *sysModel-wf-Model-SporadicComp*: *wf-Model-SporadicComp sysModel*
  by (*simp add*: *wf-Model-SporadicComp-def*)

end

Chapter 9

Initial State Examples

This chapter includes examples of how the HAMR model-driven development tool chain generates runtime state information into the Isabelle AADL-HSM state representation. This information is derived directly from and is traceable to the representation of AADL model information and state information in HAMR-generated code.

9.1 Temperature Control Example

TempControl Initial Thread States

theory *TempControlInitialThreadStates*
  imports *Model VarState PortState ThreadState*
begin

In the current HAMR implementation, all ports have queues of size 1, and the overflow policy of all port queues is `DropEarliest`. Therefore, HAMR AADL-HSM generation defines a empty queue with these attributes. The initial state of each thread has port states set to this empty queue.

definition *empty-queue*:: *int Queue* **where** [*simp add*]:
 empty-queue = mk-empty-queue 1 DropEarliest

The following definitions are generated by HAMR to represent the initial state of the `TempSensor` thread.

Local thread variable state: The `TempSensor` thread has no variables, so the *VarState* is an empty map.

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-tvar-initial* :: *int VarState* **where** [*simp add*]:
  TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-tvar-initial = Map.empty

Infrastructure Input Port State: The `TempSensor` thread has no input ports, the infrastructure input port state *infi* is an empty map.

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-infi-initial* :: *int PortState* **where** [*simp add*]:
  TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-infi-initial = Map.empty

Application Input Port State: The `TempSensor` thread has no input ports, the application input port state *appi* is an empty map.

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-appi-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-appi-initial = *Map.empty*

Application Output Port State: The `TempSensor` thread has two output ports `currentTemp` and `tempChanged`, so the *PortIds* for those ports are mapped to empty queues.

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-appo-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-appo-initial = *map-of* [((*PortId 0*), *empty-queue*),((*PortId 1*), *empty-queue*)]

Infrastructure Output Port State: The `TempSensor` thread has two output ports `currentTemp` and `tempChanged`, so the *PortIds* for those ports are mapped to empty queues.

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-info-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-info-initial = *map-of* [((*PortId 0*), *empty-queue*),((*PortId 1*), *empty-queue*)]

The *DispatchStatus* for the thread is set to *NotEnabled*.

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-disp-initial* :: *DispatchStatus* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-disp-initial = *NotEnabled*

The values above are combined into a record to form the initial thread state for `TempSensor` thread.

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-initial* :: *int ThreadState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-initial =
(*tstate*
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-infi-initial
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-appi-initial
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-appo-initial
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-info-initial
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-tvar-initial
TempControlSoftwareSystem-s-Instance-tcproc-tempSensor-disp-initial
)

State definitions for the remaining components are similar.

definition *TempControlSoftwareSystem-s-Instance-tcproc-fan-tvar-initial* :: *int VarState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-fan-tvar-initial = *Map.empty*

definition *TempControlSoftwareSystem-s-Instance-tcproc-fan-infi-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-fan-infi-initial = *map-of* [((*PortId 2*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-fan-appi-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-fan-appi-initial = *map-of* [((*PortId 2*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-fan-appo-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-fan-appo-initial = *map-of* [((*PortId 3*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-fan-info-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-fan-info-initial = *map-of* [((*PortId 3*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-fan-disp-initial* :: *DispatchStatus* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-fan-disp-initial = *NotEnabled*

definition *TempControlSoftwareSystem-s-Instance-tcproc-fan-initial* :: *int ThreadState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-fan-initial =
(*tstate*
TempControlSoftwareSystem-s-Instance-tcproc-fan-infi-initial
TempControlSoftwareSystem-s-Instance-tcproc-fan-appi-initial
TempControlSoftwareSystem-s-Instance-tcproc-fan-appo-initial
TempControlSoftwareSystem-s-Instance-tcproc-fan-info-initial
TempControlSoftwareSystem-s-Instance-tcproc-fan-tvar-initial
TempControlSoftwareSystem-s-Instance-tcproc-fan-disp-initial
)

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-tvar-initial* :: *int VarState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-tvar-initial = *Map.empty*

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-infi-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-infi-initial = *map-of* [((*PortId 4*), *empty-queue*),((*PortId 5*), *empty-queue*),((*PortId 6*), *empty-queue*),((*PortId 8*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-appi-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-appi-initial = *map-of* [((*PortId 4*), *empty-queue*),((*PortId 5*), *empty-queue*),((*PortId 6*), *empty-queue*),((*PortId 8*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-appo-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-appo-initial = *map-of* [((*PortId 7*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-info-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-info-initial = *map-of* [((*PortId 7*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-disp-initial* :: *DispatchStatus* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-disp-initial = *NotEnabled*

definition *TempControlSoftwareSystem-s-Instance-tcproc-tempControl-initial* :: *int ThreadState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-initial =

(*tstate*
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-infi-initial
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-appi-initial
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-appo-initial
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-info-initial
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-tvar-initial
TempControlSoftwareSystem-s-Instance-tcproc-tempControl-disp-initial
)

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-tvar-initial* :: *int VarState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-tvar-initial = *Map.empty*

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-infi-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-infi-initial = *map-of* [((*PortId 9*), *empty-queue*),((*PortId 11*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-appi-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-appi-initial = *map-of* [((*PortId 9*), *empty-queue*),((*PortId 11*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-appo-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-appo-initial = *map-of* [((*PortId 10*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-info-initial* :: *int PortState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-info-initial = *map-of* [((*PortId 10*), *empty-queue*)]

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-disp-initial* :: *DispatchStatus* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-disp-initial = *NotEnabled*

definition *TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-initial* :: *int ThreadState* **where** [*simp add*]:
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-initial =
(*tstate*
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-infi-initial
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-appi-initial
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-appo-initial
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-info-initial
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-tvar-initial
TempControlSoftwareSystem-s-Instance-tcproc-operatorInterface-disp-initial
)

end