System verification for AADL-based systems

by

Jacob Legg

B.S., Kansas State University, 2023

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2026

Approved by:

Major Professor
Dr. John Hatcliff

Copyright

Abstract

The AADL HAMR Semantics Mechanization (AADL-HSM) is an Isabelle-based mechanization of the informal Architecture Analysis and Design Language (AADL) semantics that formalizes the structure and execution of AADL for the High-Assurance Model-based Rapid engineering for embedded systems (HAMR) toolkit. In addition, the AADL-HSM introduces the formalization of a property framework to support component-oriented contract specification and verification. However, the existing framework does not support system-level specification and verification.

This report presents a sound system-level contract framework for an extension of the simplified variant of the AADL-HSM. The approach requires a workflow net representation of the scheduling constraints for a system to be constructed, which is then used to express assertions that should hold between the execution of two explicitly constrained disjoint sets of components in a constraint-compliant static schedule. From these assertions, along with the component-level contracts, verification conditions and independence requirements are systematically generated and then discharged via automatic methods such as SMT solvers. This approach provides a sound theoretical basis for the implementation of the framework into the full HAMR toolkit.

Table of Contents

List of Figures vii

List of Tables viii

Acknowledgements ix

1 Introduction 1

2 Background and Related Work 4

2.1 Petri Nets 4

2.1.1 Representation 5

2.1.2 Workflow Nets 7

2.2 HAMR Semantics 9

2.2.1 HAMR-Micro Semantics 10

3 HAMR-Micro Extension 16

3.1 Changes to Scheduling 16

3.1.1 Schedule State Update 17

3.1.2 Scheduling Constraint Structures 18

3.1.3 Well-Formedness Conditions 21

3.1.4 Example 22

3.2 Changes to the Representation and Semantics 23

3.2.1 Model 23

3.2.2 System 24

3.2.3 System State 26

3.2.4 Execution 27

4 System Specification 29

4.1 Component Contracts 29

4.2 Component Write Frames 30

4.3 System Contracts 32

4.3.1 Motivation 32

4.3.2 Specification of the M1 System 33

4.4 Representations 35

4.4.1 Graphical Representation 35

4.4.2 Sets-and-Maps Representation 36

4.4.3 Textual Representation 37

5 Verification Conditions and Independence Requirements 41

5.1 Component Contract Verification Condition 41

5.2 System Contract Verification Condition 42

5.2.1 Initial State Verification Condition 42

5.2.2 Pre-Assert Verification Condition 43

5.2.3 Next-Assert Verification Condition 44

5.2.4 Post-Pre Verification Condition 46

5.3 Independence 47

5.3.1 Non-Blocking Requirement 48

5.3.2 Preservation of Post-Assertions 49

5.4 Soundness 50

5.4.1 Component Transitions 52

5.4.2 Control Point Transitions 57

6 Conclusions 62

6.1 Future Work 63

Bibliography 65

A WF Conditions For the Next-Relation 68

B M1 System Artifacts 70

List of Figures

2.1 Simple example of a Petri net 5

2.2 M1 example system AADL diagram 10

3.1 Petri net representation of the M1 scheduling constraints 19

3.2 Sequent transition 20

3.3 Split-Join pair 20

4.1 Constraint-based schedule-point view 34

4.2 Older schedule-point view 35

4.3 Petri-net-derived view 36

5.1 M1 Initial State VC 42

5.2 M1 AC2 Pre-Assert VC 43

5.3 M1 JOIN Next-Assert VC 44

5.4 M1 AC2 Next-Assert VC 45

5.5 M1 Post-Pre VC 46

5.6 Component case soundness diagram 53

5.7 Component case soundness diagram: Choice 54

5.8 Component case soundness diagram: Alt-Choices 55

5.9 Component case soundness diagram: JoinNotSat 56

5.10 Control point case soundness diagram 58

5.11 Control point case soundness diagram: END Choice 59

5.12 Control point case soundness diagram: Not END Choice 60

5.13 Control point case soundness diagram: Not Choice 61

List of Tables

3.1 New well-formedness conditions for the model 24

3.2 New well-formedness conditions for the system 25

3.3 New well-formedness conditions for a system state 26

4.1 Well-formedness conditions for component contracts 30

4.2 Well-formedness conditions for component write frames 31

4.3 Well-formedness conditions for system contracts 32

5.1 Inductive step cases 51

A.1 Well-formedness conditions for the next-relation 68

A.2 Well-formedness conditions for the next-relation cont. 69

B.1 M1 system artifacts 70

B.2 M1 system artifacts cont. 71

Acknowledgments

I would first like to thank Dr. John Hatcliff for being my major professor during my time as a masters student, proposing and advising this report, and supporting me and my endeavors for several years. I would also like to thank Dr. Torben Amtoft and Dr. Robby for taking the time to be members of the committee for this report.

I would also like to extend special thank yous to Timothy Tucker for being an amazing rubber duck throughout my time working on this project, to Supriya Bolla and Chris Loura for making sure I take breaks from time to time, and to all my friends over the years in the department. You guys are the best, and I couldn't have done it without you.

Finally, I would like to give a big thank you to my family for supporting me throughout my time at Kansas State University.

Chapter 1

Introduction

Model-based development (MBD) approaches can be an integral part of the development of high-assurance and safety-critical systems. MBD can provide automated code generation, testing, and verification capabilities for components and systems. In frameworks such as the High-Assurance Model-based Rapid engineering for embedded systems (HAMR) toolkit[1], component-level *contracts* play a central role in specifying desired behaviors.

For system-level reasoning, several contract-based reasoning frameworks have been developed, such as AGREE[2] and CoCoSim[3]. However, these approaches rely on synchronous dataflow execution semantics in which every component is executed in lockstep as one single step. While these approaches are useful, they do not align well with common notations of scheduling for real-time systems and industry standards such as ARINC 653[4]. In more common approaches to scheduling, a specification and verification framework needs the ability to express requirements and observe the state of the system at intermediate points in the execution of a system and reason about the ordering and composability of components.

In recent DARPA research projects focused on model-based development for safety-critical systems running on seL4, where simple static cyclic scheduling, similar to that used in ARINC 653, has been used. In these systems, task actions are executed sequentially in the order specified in a developer-supplied static schedule.

A formalization of this notion of execution was proposed by Hallerstede and Hatcliff in *A mechanized semantics for component-based systems*[5]. Their work introduced the AADL HAMR Semantics Mechanization (AADL-HSM), which is an Isabelle-based mechanization of the informal Architecture Analysis and Design Language (AADL)[6] semantics that formalizes the structure and execution of AADL for the HAMR toolkit. In this formalism, an execution step does not simulate an entire execution cycle, but instead, it executes a single component to completion and then progresses the schedule by one slot.

Based on this notion of execution semantics, they proposed a simple system-level reasoning framework that would establish a system invariant that would then be proven to hold at the start of execution and then after each execution step. While this proposal is an important step in expressing system requirements for statically scheduled systems, it is still limited in the ability to express distinct requirements of the system state at different points in the system's execution cycle. Moreover, it provides no way for system properties to be deduced from component contracts.

This report extends the approach proposed by Hallerstede and Hatcliff by providing a sound theoretical foundation for a system-level reasoning framework for statically scheduled systems using a simplified and idealized variant of the AADL-HSM for AADL-aligned model-driven development in the HAMR toolkit.

The specific contributions of this report are as follows.

- We introduce a flexible approach to specifying static scheduling requirements that represents all constraint-compliant schedules simultaneously using preexisting formalisms that non-experts have found useful via *workflow nets*.
- We design a framework to support system-level reasoning that is based on the contracts of components and additional "system assertions" that developers can add to the schedule specifications.

- We design a framework for generating verification conditions that can be checked by existing SMT-based tools.
- We prove the soundness of the verification condition generation in Isabelle.

The framework and formalism operate under the following assumptions.

- All systems considered by the framework use static cyclic scheduling.
- Each component executes to completion within its allotted time slot in the schedule.
- The framework is formalized using a simplified variant of the HAMR semantics that properly abstracts away several features of the full semantics.
- Reasoning over an executable representation of the scheduling constraints is equivalent to reasoning over the set of all valid schedules.
  - The formalism is currently unable to derive the set of all valid schedules from a representation of the constraints. Therefore, in the formalism, reasoning is done using the executable representation of the constraints.

Given this, the report is organized as follows.

- Chapter 2 summarizes the key concepts of Petri nets and workflow nets and introduces the AADL-HSM and its simplified variant, HAMR-Micro.
- Chapter 3 presents the extended definition of scheduling using workflow nets and describes the resulting changes to the structure and execution of HAMR-Micro.
- Chapter 4 presents a simplified definition of the component contracts from HAMR and extends the definition of component specification by introducing component write frames. It also outlines the definition of a system contract and its representations.
- Chapter 5 presents the verification conditions and outlines the soundness proof for the framework.
- Chapter 6 summarizes the work presented in this report and discusses the future work.

The Isabelle formalization can be found here[7].

Chapter 2

Background and Related Work

2.1 Petri Nets

A Petri net is a non-deterministic mathematical model used for the description and analysis of distributed systems established by Carl Petri in 1962[8;9]. A Petri net consists of four elements:

1. places (circles)
2. transitions (rectangles)
3. arcs (arrows)
4. tokens (dots).

Arcs go from places to transitions and vice versa. They describe the flow of information and control across the network. If an arc flows from a place to a transition, then the place is an input (*in-place*) for the transition, and if an arc flows from a transition to a place, then the place is an output (*out-place*) of the transition.

Each transition can have several inputs and outputs, where, if a token is in all input places (enabled), then the transition can be fired, where one token is removed from each input place, and one token is added to each output place.

A marking of a Petri net denotes which places have tokens in them and how many tokens each place contains.

Figure 2.1 shows a simple Petri net with an enabled transition and the result of firing the transition.

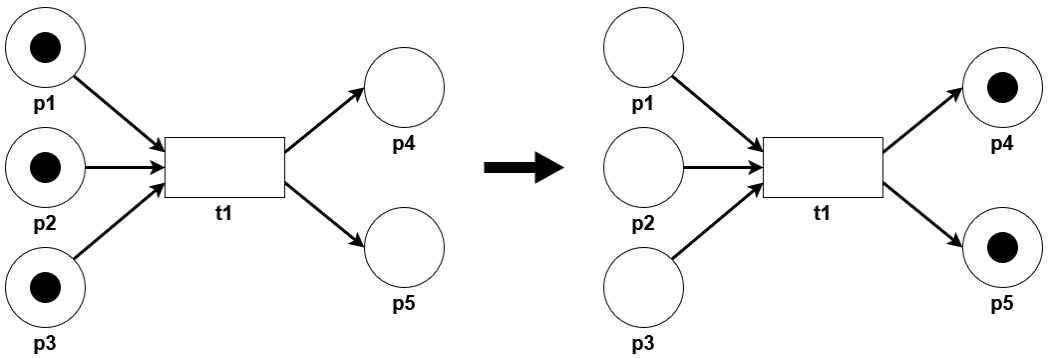

Figure 2.1: *Simple example of a Petri net*

2.1.1 Representation

For the purpose of this report, we represent the structure of Petri nets as a mapping from the set of in-places to the set of out-places for all transitions. This representation, the notation for firing sequences, and the definition of well-formedness are based on the definitions and notations for Petri nets used by Aalst in *Structural characterizations of sound workflow nets*[10]. The map representation of a Petri net is referred to as a *next-relation (NextRel)* and is denoted as

```
NextRel = {
    ({[input places of t1]} |-> {[output places of t1]}),
    ({[input places of t2]} |-> {[output places of t2]}),
    ...
    ({[input places of tn]} |-> {[output places of tn]})
}
```

For a transition `T` we define:

- In(`T`): the set of in-places
- Out(`T`): the set of out-places

While a marking of a Petri net is typically defined as a multiset representing the number of tokens in each place, for the purposes of this report, markings are instead represented as the set of places containing a token, as justified in Section 2.1.2. To distinguish between a set of places and a marking set, sets of places will be denoted by { }, and marking sets by {| |}. Accordingly, a marking is denoted as

$$\{|p1, p2, ..., pn|\}$$

A transition `T` is enabled if and only if:

$$In(T) \subseteq \{|p1, p2, ..., pn|\}$$

Firing a transition `T` on marking `M` to produce a marking `M'` is denoted as

$$M' = M - \ In(T) \ \cup \ NextRel(In(T))$$

The example demonstrated in Figure 2.1 can be represented in the new formalism as

Next-Relation (Petri Net):

$$\text{NextRel} = \{ \ \{\text{p1, p2, p3}\} \mapsto \{\text{p4, p5}\} \ \}$$

Initial Marking: {|p1, p2, p3|}

Firing t1: {|p4, p5|} = {|p1, p2, p3|} - {|p1, p2, p3|} ∪ {|p4, p5|}

Final Marking: {|p4, p5|}

For a marking M_1, the following notation can be used to express the relation between M_1 and some other marking with respect to some transition or some sequence of transitions.

- $M_1 \xrightarrow{T} M_2$: transition T is enabled in marking M_1, and firing T on M_1 results in a marking M_2
- $M_1 \rightarrow M_2$: there exists a transition T such that $M_1 \xrightarrow{T} M_2$
- $M_1 \xrightarrow{\sigma} M_n$: the firing sequence $\sigma = T_1T_2...T_{n-1}$ leads from state M_1 to state M_n

A marking M_n is reachable from M_1 ($M_1 \xrightarrow{*} M_n$) if and only if there is a firing sequence σ such that ($M_1 \xrightarrow{\sigma} M_n$). Note that the empty firing sequence is also allowed, i.e., ($M_1 \xrightarrow{*} M_1$).

Given all the above definitions, a Petri net, `PN`, is well-formed if there exists a marking `M` that satisfies the following conditions:

1. **Live:** For all markings `M'`, reachable from `M`, and all transitions `T`, there exists a marking `M''` reachable from `M'` which enables `T`

2. **Bounded:** For every marking reachable from `M` and every place `p`, the number of tokens in `p` has a finite upper bound

2.1.2 Workflow Nets

Workflow nets (WF-nets) are a special class of Petri net defined by W.M.P. van der Aalst[10;11] to formalize workflow management systems as a mathematical model that can be analyzed for special properties. Workflow management systems are generally used for the modeling, analysis, enactment, and coordination of structured business processes by groups of people, which express the general ordering in which tasks need to be completed and what tasks are required to be completed before another task can commence with a definite start and end.

Definition

A Petri net `PN` is a WF-net if and only if:

1. Petri net has two special places: START and END. Place START is a source place (no incoming arcs), and END is a sink place (no outgoing arcs)

2. If we add a transition t* to `PN`, which connects place END with START (i.e., the WF-Net is short-circuited), then the resulting Petri net is strongly connected

Free-Choice

A workflow net is free-choice if, for any two transitions that share in-places, the two transitions have equivalent sets of in-places. Given two transition T_1 and T_2, the formal definition of free-choice is

$$In(T_1) \cap In(T_2) \neq \emptyset \Rightarrow In(T_1) = In(T_2)$$

This free-choice property is meant to reduce potential conflicts between tasks, which may be influenced by the order in which preceding tasks are executed.

Soundness

A WF-net PN is sound if it satisfies the following properties:

- **Option to Complete:** For every state M reachable from state $\{|START|\}$, there exists a firing sequence leading from state M to state $\{|END|\}$. Formally

$$\forall M.\ (\{|START|\} \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} \{|END|\})$$

- **Proper Completion:** $\{|END|\}$ is the only state reachable from $\{|START|\}$ with at least one token in place END. Formally:

$$\forall M.\ (\{|START|\} \xrightarrow{*} M \ \wedge \ END \in M) \Rightarrow (M = \{|END|\})$$

- **No Dead Transitions:** There are no dead transitions in PN. Formally, for all transition T:

$$\exists M, M'.\ \{|START|\} \xrightarrow{*} M \xrightarrow{T} M'$$

According to Theorem 1 by Aalst[10], a necessary and sufficient condition for the soundness of a WF-net is that, given a short-circuited WF-Net $\overline{PN}$, the initial marking, $\{|START|\}$, of $\overline{PN}$ is live and bounded. Furthermore, given that a WF-Net is free-choice, this condition can be checked in polynomial time using algorithms derived from Rank Theorem[12] by Corollary 1 from Aalst[10]

A sound workflow net contains several desirable properties, such as a lack of deadlock and livelock for the short-circuited Petri net, and if a WF-net is free choice and sound, then the WF-net is safe (i.e., the number of tokens in a place does not exceed one) by Lemma 4 from Aalst[10]. Moreover, a sound free-choice WF-net can also be composed with other sound free-choice WF-nets to produce a sound free-choice WF-net by Theorem 3 from Aalst[10].

Due to well-formedness conditions for the next-relations and the underlying Petri nets used in this report, which will be outlined in Section 3.1.3, all Petri nets used throughout the rest of the report are sound free-choice WF-nets, and therefore markings can be defined as the set of places containing a token because the number of tokens in any place will not exceed one.

2.2 HAMR Semantics

AADL[6] is a system modeling language that contains different categories of nested components (e.g., systems, processes, and threads) to describe the structure of a system that enables early analyses of a system's architecture through an extendable notation, a tool framework, and precisely defined semantics.

The AADL HAMR Semantics Mechanization[5] (AADL-HSM), also referred to as the HAMR semantics, is an Isabelle-based mechanization of the informal AADL semantics that formalizes several aspects of an AADL model, such as ports, threads, system states, AADL runtime services, port-based communication properties, thread execution, scheduling, etc. The semantics also introduce the formalization of a property framework that can be used to reason about threads and system properties, which can lay the groundwork for contract and testing frameworks, such as the GUMBO framework[13;14] or future additions to the HAMR toolchain. The HAMR semantics also has the purpose of enabling formal reasoning between HAMR and other Isabelle-verified systems, such as the seL4 microkernel.

Due to the complexity of prototyping new systems on the full HAMR semantics, a simplified definition of the semantics was created, called the HAMR-Micro semantics, which will be described in detail in the following subsection and used throughout the rest of the report.

2.2.1 HAMR-Micro Semantics

The HAMR-Micro semantics[15] is an Isabelle-based simplification of the HAMR semantics developed by Hatcliff for prototyping new HAMR semantics concepts for the HAMR framework, such as the system-level contracts discussed in this report. As the notions of execution are determined early by thread components in the current version of HAMR, the HAMR-Micro semantics follows the approach in AADL-HSM and represents only the semantics of thread components. In the following subsections, it will be described how the HAMR-Micro semantics simplifies certain aspects of the HAMR semantics in both the representation of the system and models and the definition of execution, communication, and state.

In this section, we also introduce the M1 example system that will be used as a running example for the rest of the report. The M1 system is a simple system that continuously flips values of the a1 and b1 channels, where the values are never equal. The model of this system can be seen in Figure 2.2. In the following sections, the HAMR-Micro semantics description of the M1 example will be presented alongside the new definitions. Certain details will be omitted for now and will be described later. The location of all artifacts and details for the M1 system can be found in Appendix B.

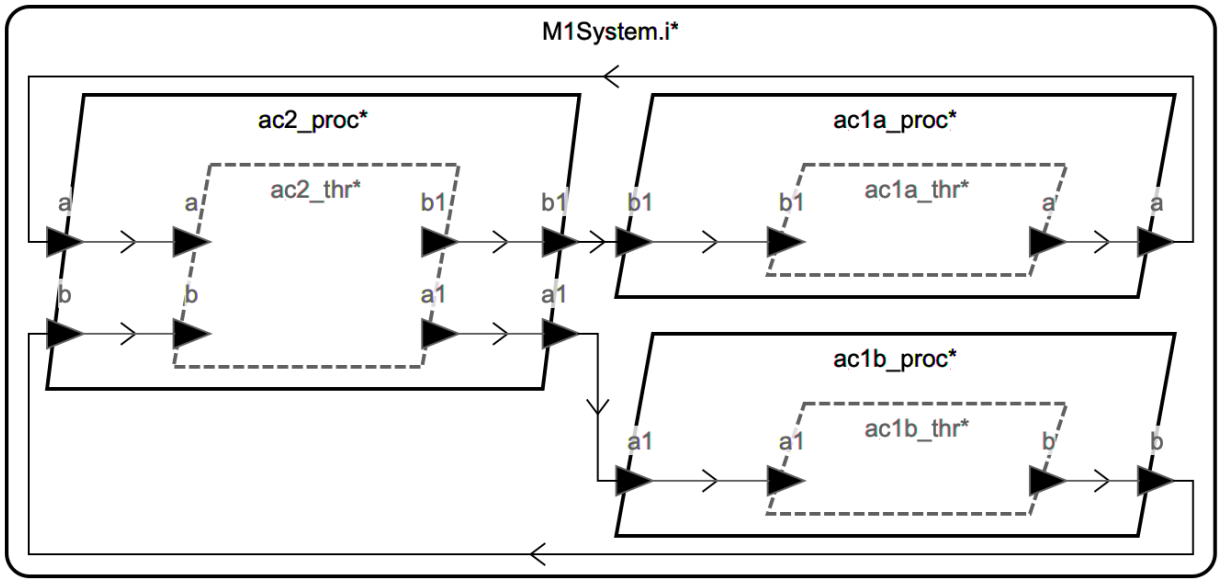

Figure 2.2: *M1 example system AADL diagram*

Model

The HAMR-Micro instance models simplify several aspects of the HAMR model, such as the definitions of component descriptions (tasks/components are the terminology used for threads in HAMR-Micro) and the topology and definition of communication. The description of a component is simplified to only refer to the in and out channels of the component, along with the state variables. Channels are a simplification of the port/communication semantics of HAMR, as they represent a single-element container that stores a value that can be read from any component that has the channel as an `in` channel and written to by any component that has the channel as an `out` channel. With this change, the channels are similar to data ports in the original HAMR semantics. Consequently, with the switch to channels, the instance model no longer records port descriptions and instead keeps track of all channel IDs, as the channel semantics have a fixed definition. This reduces the definition of a model to

```
record TaskDescr =
  inChIds :: "ChIds"
  outChIds :: "ChIds"
  varIds :: "VarIds"

record Model =
  modelTaskDescrs :: "(Tid, TaskDescr) map"
  modelChIds :: "ChIds"
```

The model for M1 is

```
definition M1Model :: "Model"
  where "M1Model = (|
          modelTaskDescrs = map_of [(TID_AC1A, AC1A_Descr),
                                    (TID_AC1B, AC1B_Descr),
                                    (TID_AC2, AC2_Descr)],
          modelChIds = {''a'', ''b'', ''a1'', ''b1''}
    |)"
```

Scheduling

Scheduling for HAMR-Micro is significantly simplified, as the components are executed to completion as they appear in a user-defined static schedule, and, therefore, they no longer have a life cycle. We therefore represent the schedule as a list of component IDs and the schedule state as a record containing the ID of the last executed component (currTid) and the slot number of the next component in the schedule to be executed (nextSlotNum). This definition of schedule and schedule state is therefore changed to

```
type_synonym Schedule = "Tid list"

record ScheduleState =
  currTid :: Tid
  nextSlotNum :: SlotNum
```

An example schedule for M1 is

```
definition M1Schedule :: "Schedule"
    where M1Schedule = [TID_AC1A, TID_AC1B, TID_AC2]
```

Progressing the schedule is defined as a function (scheduleNext) that constructs a new schedule state by setting currTid to the task ID in the next schedule slot (fetchTidFromSlot) and then setting nextSlotNum to the next slot number in the schedule (computeNextSlotNum). Progressing the schedule is therefore defined as

```
// Move back the slot number one position, and wrap around
fun computeNextSlotNum :: "Schedule => SlotNum => SlotNum"
  where "computeNextSlotNum sch slotNum =
           (if (slotNum < length sch - 1)
            then slotNum + 1
            else 0)"

fun scheduleNext :: "Schedule => ScheduleState => ScheduleState"
  where
    "scheduleNext sch schst =
      mkScheduleState (fetchTidFromSlot sch (nextSlotNum schst))
                      (computeNextSlotNum sch (nextSlotNum schst))"
```

Note: The currTid in the initial schedule state is 0 to represent the AADL initialization phase and is therefore not tied to a component.

System State

The definition of state for the HAMR-Micro representation is simplified to only maintain the component states, the channel states, the schedule state, and the previous component and channel states. Component states are simplified to only maintain the value of local variables (tvar), as there is no notion of ports, and the channel states maintain the value stored on each channel. The schedule state now refers to the new definition of schedule state defined in the previous section. The definition of system state is therefore reduced to

```
type_synonym 'a VarState = "(VarId, 'a) map"

record 'a TaskState =
  tvar :: "'a VarState"

type_synonym 'a ChState = "(ChId, 'a) map"

record 'a SystemState =
  systemTasks :: "(Tid, 'a TaskState) map"
  systemChs :: "'a ChState"
  scheduleState :: ScheduleState
  previousSystemTasks :: "(Tid, 'a TaskState) map"
  previousSystemChs :: "'a ChState"
```

The initial system state for M1 would be

```
definition M1InitSystemState :: "int SystemState"
  where "M1InitSystemState = (|
          systemTasks = map_of [(TID_AC1A, initTaskState_AC1A),
                                (TID_AC1B, initTaskState_AC1B),
                                (TID_AC2, initTaskState_AC2)],
          systemChs = M1InitChState,
          scheduleState = (| currTid = 0, nextSlotNum = 0 |),
          previousSystemTasks = map_of [(TID_AC1A, initTaskState_AC1A),
                                        (TID_AC1B, initTaskState_AC1B),
                                        (TID_AC2, initTaskState_AC2)],
          previousSystemChs = M1InitChState
        |)"
```

System

With HAMR-Micro, an explicit definition of a system is introduced, which holds the initial state of the system, a map from component IDs to their corresponding component definitions (task map), and the static schedule of the system. A task definition acts similarly to the AppBehaviors in the HAMR semantics, but, instead of providing a constraint on the behavior of a component, the task definitions provide an explicit definition of the implementation (action), which will later be checked if it conforms to the expected behavior via the component contracts in Section 4.1. The formal definition of a system is

```
type_synonym 'value Action =
  "('value TaskState × 'value ChState)
   => ('value TaskState × 'value ChState)"

record 'value Task =
  taskId :: Tid
  action :: "'value Action"

type_synonym 'value TaskMap = "(Tid, 'value Task) map"

record 'value System =
  initSystemState :: "'value SystemState"
  taskMap :: "'value TaskMap"
  schedule :: Schedule
```

The system for the M1 is

```
definition M1TaskMap :: "int TaskMap"
  where "M1TaskMap = map_of [(TID_AC1A, Task_AC1A),
                             (TID_AC1B, Task_AC1B),
                             (TID_AC2, Task_AC2)]"

definition M1System :: "int System"
    where M1System = (|
        initSystemState = M1InitSystemState,
        taskMap = ,
        schedule = M1Schedule
    |)
```

Execution

The definition of an execution step (execution semantics) for the HAMR-Micro semantics changes significantly from the original HAMR semantics definition as a result of all the changes stated above. With all the above-stated definitions, the definition of execution has been simplified to

1. Progress the schedule state (scheduleNext) based on the schedule
2. Obtain the action tied to the current component ID of the schedule state
3. Apply the action to the local state of the task/component
4. Construct the updated system state using the updated local state and updated schedule state

Chapter 3

HAMR-Micro Extension

3.1 Changes to Scheduling

For this report, we extend the definition of scheduling in the HAMR-Micro semantics by replacing the static schedule with a *next-relation*, which represents the scheduling constraints for a single schedule cycle. This next-relation should define a partial ordering of the components that expresses strict precedence constraints between pairs of components (i.e., a component must come before a specific component), along with what sets of components have no constraints between them yet have common constraints on what can precede and succeed each set.

To support this, the next-relation must represent a sound, acyclic, free-choice WF-net (AFC-WF-net), which satisfies several side conditions that restrict what constraints are expressible by the WF-net. The motivation of this requirement is that WF-nets are a user-friendly formalism for non-experts that's designed to capture the constraints on the ordering of tasks in a workflow, making them suitable for capturing the scheduling constraints of components in a schedule. Moreover, due to several properties of AFC-WF-nets and constraints on a next-relation's domain, the underlying Petri net characterizes the set of all valid schedule bodies as the set of firing sequences from $\{|START|\}$ to $\{|END|\}$.

The following subsections will introduce the updated definition of a schedule state update, the necessary structural patterns and transition categories to express the constraints, and the well-formedness conditions for the next-relation-based scheduling.

3.1.1 Schedule State Update

With the change to next-relation-based scheduling, the definition of updating the schedule state (scheduleNext) also changes. The updated definition consists of two cases:

1. If the execution of the underlying Petri net is completed (i.e., the schedule state is $\{|END|\}$), reset the schedule state to $\{|START|\}$

2. Otherwise, fire an enabled transition following the definition in Section 2.1

These two cases exist because the execution of the next-relation corresponds to the execution of a single schedule cycle. The addition of the reset case allows for repeated execution of the next-relation. Additionally, by establishing this separation, reasoning about the schedule can be explicitly separated into (a) reasoning about a single schedule cycle and (b) reasoning about the indefinite behavior of the schedule.

With these changes, the formal definition of updating the schedule state for the extended HAMR-Micro semantics is

```
fun scheduleNextBody :: "Next => Ready => Rid fset => Ready"
  where "scheduleNextBody n r rf =
    r - (rf :: nat fset) |\cup| n $ rf"

fun scheduleNextTotal :: "Next => Model => Ready => Rid fset => Ready"
  where "scheduleNextTotal next m r rf =
    (if (rf = {|modelEndRid m|})
        then {|modelStartRid m|}
        else scheduleNextBody next r rf)"
```

Note: \cup is equivalent to the $\cup$ symbol.

In this definition, the first function `scheduleNextBody` takes in a next-relation `n`, a schedule state `r`, and the set of in-places for a transition `rf` as arguments. The function then calculates the new schedule state by removing the set of in-places from the schedule state and then adding the set of corresponding out-places (i.e., r - rf |∪| n $ rf), where $ is an operator that applies the next-relation to a set of in-places for a transition and returns the corresponding out-places. This function directly corresponds to the definition of firing a transition that was presented in Section 2.1.

The second function `schedulNextTotal` takes in a next-relation `n`, a model `m`, a schedule state `r`, and the set of in-places for an enabled transition `rf` as arguments. The function then checks if the schedule state is being reset (i.e., $rf = \{|modelEndRid\ \ m|\}$) and updates the schedule state accordingly. In the *then* branch of this function, the schedule state $\{|START|\}$ is returned, which corresponds to case one. In the *else* branch, `scheduleNextBody` is called with `n`, `r`, and `rf` and returns the resulting state, which corresponds to case two.

3.1.2 Scheduling Constraint Structures

In order to ensure that the next-relation and its underlying Petri net express the desired types of constraints, we introduce several categories of transitions that all transitions must belong to and specify several structural patterns to represent constraints.

All transitions within the Petri net underlying the next-relation must be either *component* transitions or *control point* transitions. Component transitions are transitions that are correlated to some component, and the firing of said transition represents the execution of the component. Examples of component transitions can be seen in Figure 3.1 with the AC1A, AC1B, and AC2 transitions, which correspond to the AC1A, AC1B, and AC2 components of Figure 2.2. Control point transitions, on the other hand, are transitions that are used to express constraints or lack thereof and are not tied to a component. Examples of control point transitions can be seen in Figure 3.1 with the SPLIT and JOIN transitions, which do not appear in Figure 2.2.

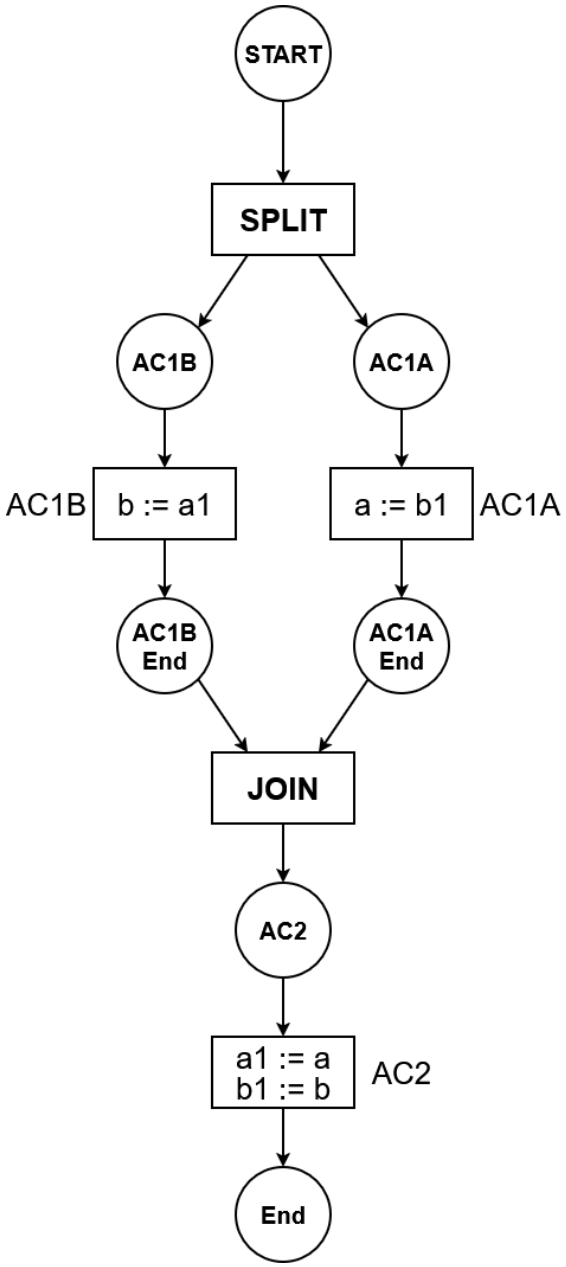

Figure 3.1: *Petri net representation of the M1 scheduling constraints*

These transitions are further differentiated into three types: sequent, split, and join. Sequents are component transitions that have one input and one output place, where a series of sequents represents a total precedence ordering for a series of components, where components that appear earlier in the series have higher precedences (i.e., are executed earlier). A graphical representation of a sequent can be seen in Figure 3.2.

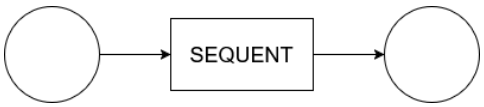

Figure 3.2: *Sequent transition*

Splits are control point transitions that have one in-place and several out-places, and joins are control point transitions with several in-places and one out-place. In a well-formed next-relation for this framework, each split must have a corresponding join where each out-place of the split has a path to an in-place of the join, where each path is disjoint and has at least one sequent in the underlying Petri net. This requirement establishes a structure, which will be referred to as a *split-join pair*.

Within a split-join pair, each path can be interpreted as a set of components with their own local scheduling constraints. Consequently, a split-join pair represents when several sets of components have their own local constraints but have common global constraints on what precedes and succeeds all sets. The body of a split-join pair can therefore be informally thought of as separate sets of components that are to run concurrently. A graphical representation of a split-join pair can be seen in Figure 3.3.

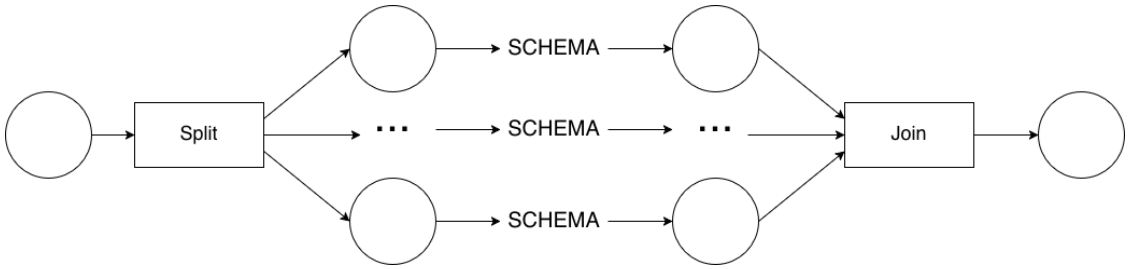

Figure 3.3: *Split-Join pair*

By combining the concepts of sequents, series of sequents, and split-join pairs, we define a structure called a *schedule schema*. A schedule schema is an ordered series of sequents and split-join pairs, where the components associated with each element have a higher priority than those in subsequent elements. Since a series of sequent transitions represents a strict precedence ordering and split-join pairs represent the lack of constraints between several sets of components with their own local constraints, a schedule schema represents a partial precedence ordering.

Because the Petri net underlying the next-relation can only be constructed from component transitions and control point transitions, all transitions are sequents, splits, or joins. Therefore, due to the constraints of the Petri net and its transitions, the Petri net is a schedule schema and, therefore, defines a partial precedence ordering of all components in the system, where the ordering can only express strict precedence constraints between components and the lack of constraints between sets of components with common preceding and succeeding components.

3.1.3 Well-Formedness Conditions

For a Petri net representation of the scheduling constraints to be well-formed, the next-relation and the underlying Petri net need to satisfy the conditions listed in Appendix A. In the appendix, the well-formedness conditions necessary to enforce that the Petri net is a sound AFC-WF-net are specifically labeled in the table.

Several of the listed well-formedness conditions are too complex to check within the current Isabelle framework due to the complexity of reasoning about firing sequence and reachability for a next-relation representation of a Petri net. The following conditions are therefore assumed to be true for all systems specified in Isabelle:

- wf_System_NextRel_PlaceOnPath
- wf_System_NextRel_TransitionOnPath
- wf_System_NextRel_OptToComplete
- wf_System_NextRel_ProperComplete
- wf_System_NextRel_NoDeadTransitions
- wf_System_NextRel_NoCycle
- mayHappenInParallelInd
- BackStepReachability

Despite the limitations of representation in Isabelle, several algorithms exist to check these conditions. Such algorithms could be implemented in the HAMR implementation as well-formedness checks in a post-processing phase that occurs immediately after parsing of the developer-facing representation of the Petri net. The details of these algorithms are beyond the scope of the report and will therefore be discussed in future documentation.

3.1.4 Example

As stated in the previous chapter, M1 consists of three components: AC1A, AC1B, and AC2. The scheduling of these components must satisfy the following requirements:

- AC1A must execute before AC2
- AC1B must execute before AC2

These constraints can be expressed graphically as the Petri net in Figure 3.1 or by the following next-relation.

```
M1_NextRel = {
               {|START|} |-> {|RID_AC1A, RID_AC1B|},
               {|RID_AC1A|} |-> {|RID_AC1A_End|},
               {|RID_AC1B|} |-> {|RID_AC1B_End|},
               {|RID_AC1A_End, RID_AC1B_End|} |-> {|RID_AC2|},
               {|RID_AC2|} |-> {|END|}
              }
```

The Petri net and next-relation indicate that both AC1A and AC1B must occur before AC2 in the schedule, with no ordering constraints between AC1A and AC1B. It also expresses that AC1A and AC1B have a common predecessor, being the start of the schedule (i.e., either AC1A or AC1B can be scheduled first), and that they have a common successor, being AC2.

The lack of constraints between AC1A and AC1B is expressed by the split-join pair, where both AC1A and AC1B exist on separate parallel paths between the split and join. The constraint that both AC1A and AC1B must precede AC2 is expressed by AC2 coming after the join, since the join requires that all components between the split and join be scheduled before any component following the join. Therefore, the set of all valid linear schedules represented by the Petri net and next-relation for M1 is the following:

- [AC1A, AC1B, AC2]
- [AC1B, AC1A, AC2]

3.2 Changes to the Representation and Semantics

With the introduction of the next-relation–based scheduling, several core components of the HAMR-Micro semantics, including the definition of a model, a system, system states, and execution, need to be modified to handle the non-determinism of the next-relation-based scheduling.

3.2.1 Model

The definition of a model is extended to add the set of place identifiers (ready IDs) for the next-relation alongside the ID for the START place and the END place. With this change, the formal definition of a model for the HAMR-Micro extended semantics is

```
record Model =
  modelTaskDescrs :: "(Tid, TaskDescr) map"
  modelChIds :: "ChIds"
  modelReadyIds :: "Rids" //NEW
  modelStartRid :: "Rid"  //NEW
  modelEndRid :: "Rid" //NEW
```

and the well-formedness conditions in Table 3.1 are introduced.

Table 3.1: *New well-formedness conditions for the model*

WF Condition	Description
wf_Model_Rids_1	The number of places must be greater than the number of components
wf_Model_Rids_2	START must be a member of the set of places
wf_Model_Rids_3	END must be a member of the set of places
wf_Model_Rids_4	START must not equal END

With the updated definition of a model, the definition of M1's model becomes

```
definition M1Rids :: "Rids"
    where "M1Rids =
        {|START, RID_AC1A, RID_AC1B,
          RID_AC1A_End, RID_AC1B_End, RID_AC2, END|}"


definition M1Model :: "Model"
    where "M1Model = (|
        modelTaskDescrs = map_of [(TID_AC1A, AC1A_Descr),
                                  (TID_AC1B, AC1B_Descr),
                                  (TID_AC2, AC2_Descr)],
        modelChIds = M1Chs,
        modelReadyIds = M1Rids,
        modelStartRid = START,
        modelEndRid = END
    |)"
```

3.2.2 System

The definition of a system is extended to include an activation map, a next-relation, and an initial system state. The activation map maps transitions to components by mapping the in-places of the transition to the ID of the component, thereby differentiating component transitions from control point transitions. The initial state replaces the explicit initialization phase in scheduling in order to simplify the reasoning for the prototype of the framework. Lastly, the next-relation replaces the static schedule due to the change in the definition of scheduling.

With these changes, the formal definition of a system for the HAMR-Micro extended semantics is

```
record 'value System =
  initSystemState :: "'value SystemState"
  taskMap :: "'value TaskMap"
  nextRel :: Next //NEW
  activationMap :: RidToTidMap //NEW
```

and the well-formedness conditions in Table 3.2 are introduced.

Table 3.2: *New well-formedness conditions for the system*

WF Condition	**Description**
wf_System_ActivationMap_1	All sets in the domain of the activation map must be transitions in the next-relation
wf_System_ActivationMap_2	All elements in the range of the activation map must be an ID of a component in the system
wf_System_NextRel	See Appendix A
wf_System_init	The initial schedule state must be $\{\|START\|\}$

With the updated definition of a system, the definition of M1's system becomes

```
definition M1Next :: "Next"
  where "M1Next = map_of [({|START|}, {|RID_AC1A, RID_AC1B|}),
                          ({|RID_AC1A|}, {|RID_AC1A_End|}),
                          ({|RID_AC1B|}, {|RID_AC1B_End|}),
                          ({|RID_AC1A_End, RID_AC1B_End|}, {|RID_AC2|})
                          ({|RID_AC2|}, {|END|})]"

definition M1ActivationMap :: "RidToTidMap"
  where "M1ActivationMap = map_of [({|RID_AC1A|}, TID_AC1A),
                                   ({|RID_AC1B|}, TID_AC1B),
                                   ({|RID_AC2|}, TID_AC2)]"

definition M1System :: "int System"
    where "M1System = (|
        initSystemState = M1InitSystemState,
        taskMap = M1TaskMap,
        nextRel = M1Next,
        activationMap = M1ActivationMap
    |)"
```

3.2.3 System State

The definition of a system state is modified to replace the original definition of schedule state with the ready set, which represents the marking on the underlying Petri net. With this change, the formal definition of a system state for the extended HAMR-Micro semantics is

```
record 'a SystemState =
  systemTasks :: "(Tid, 'a TaskState) map"
  systemChs :: "'a ChState"
  ready :: Ready //NEW
  previousSystemTasks :: "(Tid, 'a TaskState) map"
  previousSystemChs :: "'a ChState"
```

and the well-formedness conditions in Table 3.3 are introduced.

Table 3.3: *New well-formedness conditions for a system state*

WF Condition	**Description**
wf_SystemState_ConditionalDeadlock	The schedule state is not $\{\|END\|\}$ if and only if there exist an enabled transition
wf_SystemState_Reach	The schedule state must reachable from $\{\|START\|\}$
wf_SystemState_ReadySub	The schedule state must be a subset of the set of places of the Petri net

With the updated definition of a system state, the definition of M1's initial system state becomes

```
definition M1InitSystemState :: "int SystemState"
  where "M1InitSystemState = (|
          systemTasks = map_of [(TID_AC1A, initTaskState_AC1A),
                                (TID_AC1B, initTaskState_AC1B),
                                (TID_AC2, initTaskState_AC2)],
          systemChs = M1InitChState,
          ready = {|START|},
          previousSystemTasks = map_of [(TID_AC1A, initTaskState_AC1A),
                                        (TID_AC1B, initTaskState_AC1B),
                                        (TID_AC2, initTaskState_AC2)],
          previousSystemChs = M1InitChState
        |)"
```

3.2.4 Execution

With the changes to the representation of scheduling, the definition of execution changes significantly due to the non-determinism of the next-relation. As a result, the execution semantics must also be non-deterministic, allowing it to characterize the execution of a system with any valid schedules. Furthermore, because the next-relation contains control point transitions, which do not explicitly have an associated action, there exist two cases for the definition of execution, with one case being for component transitions and the other being for control point transitions. Therefore, the new definition of an execution step for the extended HAMR-Micro semantics is the following:

1. Non-deterministically pick an enabled transition from the ready set or $\{|END|\}$ if present

2. Determine if the transition is a component or control point transition

   - Case 1: If the chosen transition is a component transition

     (a) Update the ready set using `scheduleNextTotal`

     (b) Obtain the ID of the associated component

     (c) Obtain the action of the component

     (d) Apply the action to the local state of the component

     (e) Construct the updated system state with the updated local state of the component and the updated ready set

   - Case 2: If the chosen transition is a control point transition

     (a) Update the ready set using `scheduleNextTotal`

     (b) Construct the updated system state with the updated ready set

Note: It has been proven that this definition of an execution step produces well-formed system post-states, given a well-formed system, model, and system pre-state. This proof, along with the formal definition of the execution semantics, can be found in *HAMRMicro05ExecutionSemantics.thy* in the repository for the report[7].

Chapter 4

System Specification

4.1 Component Contracts

Component contracts for this framework are designed as a simplification of the component contracts in the full HAMR framework. This version simplifies the original by removing the contracts for the initialization phase, which itself has been removed and replaced with an explicit initial state. As a result, the component contracts now define the behavior of each component in terms of the precondition and postcondition of the action (i.e., its entry point).

Like the application behaviors of the HAMR semantics[5] and the Gumbo contracting language of the HAMR toolkit, these contracts are meant to express the properties of a component without specifying the implementation of the component. This separation creates a layer of abstraction between the definition of the system and the component implementation. Consequently, this allows for the component contract to be verified on the component level and then used for reasoning about the component at the system level without the baggage of the implementation. In the final HAMR version of the system-level reasoning framework, the component contracts will be assumed to have been verified by tools such as Logika[13;16] or Verus[17] on the component level.

A component contract is well-formed if the contract satisfies the conditions stated in Table 4.1.

Table 4.1: *Well-formedness conditions for component contracts*

WF Condition	**Description**
wf_TaskContracts_ChState_Pre	Preconditions only refer to the in channels for the task
wf_TaskContracts_ChState_Post	Postconditions only refer to the in channels of st1 and st2 and the out-channels st1' and st2'
wf_TaskContracts_1	Every component has a precondition
wf_TaskContracts_2	Every component has a postcondition

Note: A component contract for M1 will not be specified since the component contracts are not the focus of this paper. Therefore, the precondition and postcondition of a component `C` will be referred to as follows for any future M1 example.

- Pre_C: The precondition of `C`
- $Post_C$: The postcondition of `C`

4.2 Component Write Frames

Component write frames are a predicate that extends the definition of component specification by explicitly specifying what parts of the component and system state are unmodified by the execution of a component's action. These are necessary because reasoning about the behavior of a component on the system level can only be done using a component's contract. While some parts of these frame conditions can be included as part of a component's contract, (a) this is not always possible, since component contracts can only reference local channels and variables, and (b) doing so would over-encumber the component contracts with explicit frame conditions.

In the Isabelle version of the system-level contracts, these predicates must be provided and checked for each component because the actions are a shallow embedding of the implementation of a component that uses operations defined in Isabelle and therefore cannot be

parsed. However, in the HAMR version, the frames can be automatically derived from the definition of the entry point of the component using preexisting methods.

To make the notion more precise, the provided/generated frame condition can be decomposed into two separate predicates: the local and global write frames. The local frame states which local variables and out-channels of the component are unmodified by the action of the component, and the global frame states which component states and channels are unmodified by the action of the component.

The component write frames are well-formed if the frames satisfy the conditions stated in Table 4.2.

Table 4.2: *Well-formedness conditions for component write frames*

WF Condition	**Description**
wf_WriteFrame_Dom	Every component has a local and global frame
wf_WriteFrame_LocalExec	The local frame is correct for the action of the component
wf_WriteFrame_GlobalExec	The global frame is correct for the action of the component

As an example of a component write frame, the component write frame for AC2 is

```
LocalFrame_AC2 = True


GlobalFrame_AC2 = st1TID_AC1A = st1.AC1A_st = st2.AC1A_st
                        /\ st1.AC1B_st = st2.AC1B_st
                        /\ st1.a = st2.a
                        /\ st1.b = st2.b
```

Note: For the rest of the report, the local and global frames for a component `C` will be referred to as the function $LocalFrame_C$ and $GlobalFrame_C$, which will take a pre-state `st1` and post-state `st2` as arguments.

In the example, the local write frame is True as all channels and local variables are modified, whereas the global write frame expresses that the state variables of the AC1A and AC1B and the a and b channels are unaffected by the action of AC2.

4.3 System Contracts

System contracts assign system assertions for every place in the Petri net, which should be true whenever the place is in the ready set. This approach allows for assertions to be stated in terms of a set of preceding components and a set of succeeding components, where the latter must execute at a later point in a schedule cycle in accordance with the constraints. Consequently, given a system that uses a constraint-conformant schedule, the system should satisfy the system contract.

A system contract is well-formed if the contract satisfies the conditions stated in Table 4.3

Table 4.3: *Well-formedness conditions for system contracts*

WF Condition	Description
wf_SystemSpec_1	All places in the Petri net have a system assertion

Note: For the remainder of the report

- All assertions tied to the in-places of a transition are referred to as the pre-assertions
- All assertions tied to the out-places of a transition are referred to as the post-assertions

The conformance of a system to its contract will be verified by the verification conditions and independence requirements provided in Chapter 5. Representations of the system contract will be provided in Section 4.4.

4.3.1 Motivation

This approach to a system-level contract motivates the usage of workflow nets to define scheduling. From previous versions of the framework, it was discovered that in large, complex systems, it was difficult to propose strong, comprehensible assertions between the execution of any two consecutive components in a static schedule, especially when parts of the assertion must remain valid for later points in the schedule. As a result, this specification approach required a significant amount of trial and error and careful planning to ensure that relevant portions of the assertions persisted throughout execution. It also introduced a significant

amount of bloat, as assertions had to be continuously restated at each point in the schedule until they were no longer valid.

The approach presented in the report addresses these problems by deconstructing the system specifications at each point in a static schedule into smaller, more modular specifications. These specifications describe a property of all system states between the execution of two sets of components with explicit precedence constraints. As a result, the method eliminates the need for the user to manually reason about how long an assertion persists and continuously restate assertions.

Note: It has not been formally proven that all systems that use constraint-compliant schedules satisfy the system contract due to the inability to derive the set of all schedules in the Isabelle version of the framework. Addressing this limitation is a source of future work and will be discussed in Section 6.1. Nevertheless, it is reasonable to assume that this property holds for the final HAMR implementation, given the definition of execution and scheduling, the soundness proof, and the fact that the schedule conformance will be checked against the reachability graph of the Petri net (i.e., a graph representing all firing sequences of a Petri net).

4.3.2 Specification of the M1 System

Given the above definition of system contracts, and to support the requirement that the M1 system must be a system that continuously flips the values on a1 and b1 channels, where the values are never equal, the following contract is proposed using the constraints provided in Section 3.1.4:

- **INV:** The a1 channel and the b1 channel are never equal
- **REQ_1:** Between AC1A and AC2, the a channel equals the b1 channel
- **REQ_2:** Between AC1B and AC2, the b channel equals the a1 channel
- **REQ_3:** Between AC2 and the end of the schedule cycle, the a1 and a channels are equal, and the b1 and b channels are equal

These requirements could be represented graphically for a constraint-compliant schedule [AC1A, AC1B, AC2], as shown in Figure 4.1.

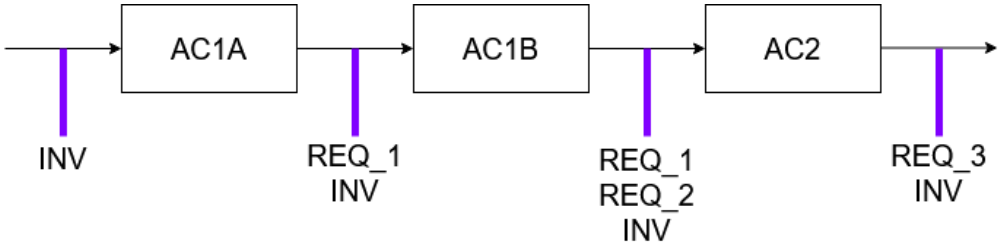

Figure 4.1: *Constraint-based schedule-point view*

To demonstrate the advantages of the new approach, an equivalent system contract is presented using the old framework, which specifies assertions at each point in a schedule. For the same schedule [AC1A, AC1B, AC2], the contract would be expressed as:

- **INV:** The a1 channel and the b1 channel are never equal
- **REQ_1:** Between AC1A and AC1B, the a channel equals the b1 channel
- **REQ_2:** Between AC1B and AC2:
  - The a channel equals the b1 channel
  - The b channel equals the a1 channel
- **REQ_3:** Between AC2 and the end of the schedule cycle
  - The a1 channel equals the a channel
  - The b1 channel equals the b channel

These requirements could be represented graphically for the schedule, as shown in Figure 4.2.

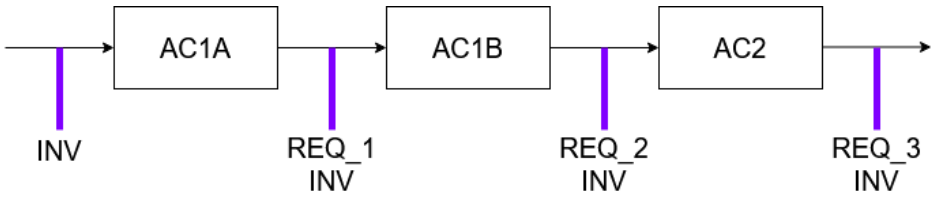

Figure 4.2: *Older schedule-point view*

Notice how both versions of the system contracts specify the same system behavior for the same schedule. However, the older approach requires the repetition of assertions (e.g., a channel equals the b1 channel) and results in denser requirements. In contrast, the newer approach separates the requirements into smaller and more modular requirements.

In larger, complex systems, the problems with the old approach would only be compounded, as more assertions would need to be restated, and consequently, requirements would expand in size. This would lead to denser and less comprehensible requirements, whereas with the newer approach, requirements would remain more concise and maintainable.

4.4 Representations

In order to work with the system-level contracts, several representations are needed: the *graphical representation*, *the sets-and-maps representation*, and *the textual representation*. The following sections will go over each in detail.

4.4.1 Graphical Representation

The graphical representation of the contract is purely for presentation. This representation is created by modifying a graphical representation of the Petri net by adding parallelograms that contain assertions about the system and assigning one to each place (circle) in the representation. An example of this can be seen in Figure 4.3 as an extension of Figure 3.1.

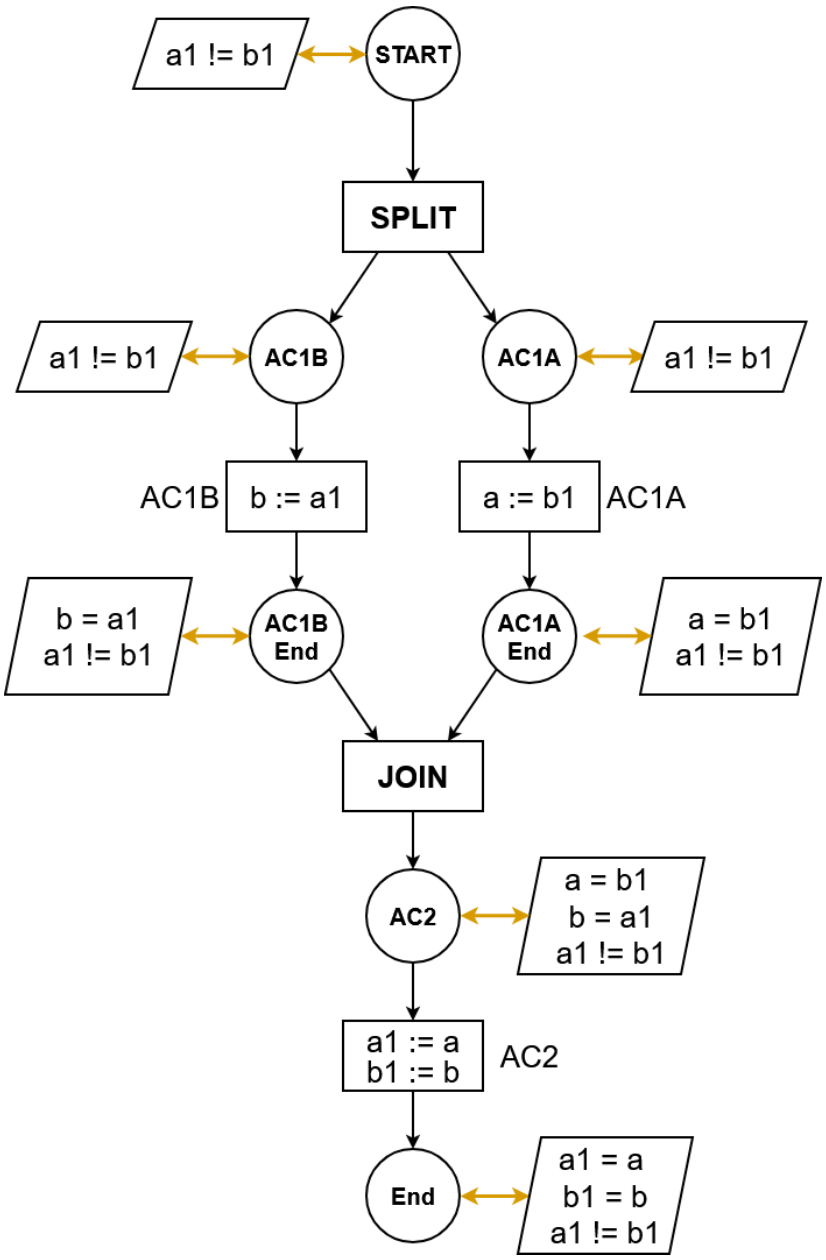

Figure 4.3: *Petri-net-derived view*

4.4.2 Sets-and-Maps Representation

The sets-and-maps representation of the system-level contracts is used to generate the verification conditions and independence requirements for the system-level contract. This representation is created by extending a next-relation with a map that maps all places of the Petri net to a system assertion. The following example demonstrates a sets-and-maps system contract for the M1 example.

```
(* Names of assertions used for brevity *)
M1_System_Spec = {START        |-> START_SysAssert
                  RID_AC1A     |-> RID_AC1A_SysAssert,
                  RID_AC1A_End |-> RID_AC1A_End_SysAssert,
                  RID_AC1B     |-> RID_AC1B_SysAssert,
                  RID_AC1B_End |-> RID_AC1B_End_SysAssert,
                  RID_AC2      |-> RID_AC2_SysAssert
                  END          |-> END_SysAssert}
```

4.4.3 Textual Representation

The textual representation of the system-level contracts is the contract that is created by the user while modeling the system. This representation is used to generate the next-relation and the corresponding contract in sets-and-maps form. This representation is also used to enforce some structural well-formedness condition of the next-relation, such as the condition that splits must match to a corresponding join.

Note: The textual representation does not appear in the Isabelle model and will only be used in the final HAMR implementation.

The following sections will go into more detail about the textual representation by presenting the components of the contract and the grammar, along with an example.

Atomics

There are two atomic elements of contracts: Components and SysAsserts. Components have one field, which is the name of some component in the model, which serves to represent a component transition in the next-relation. Components can be represented textually as

```
Component { Name: <Component Name>}
```

SysAsserts have two fields, which are the name of the assertion and the assertion itself, which serves two purposes: representing a place in the next-relation and the system assertion that is associated with the place. This can be represented textually as

```
SysAssert {
    Name: <Component Name>,
    Assert: <System Assertion>
}
```

Constraint Structures

Constraint structures are elements of the contract that are used to express the scheduling constraint structure defined in Section 3.1.2. The two constraint structures are Schemas and SplitJoins. A Schema is a series of SysAsserts, Components, and Splits that must happen in a fixed ordering that coincides with the schema scheduling constraint structure. Schemas can be represented textually as

```
Schema { ... }
```

A SplitJoin is a series of Schemas that coincide with the split-join pair scheduling constraint structure. SplitJoins can be represented textually as

```
SplitJoin {
    Schema { ... },
    Schema { ... },
    ...
}
```

Grammar

Given the above atomics and constraint structures, the EBNF grammar of the textual contract is

```
<CON> -> "Contract" <BODY>

<SCH> -> "Schema" <BODY>

<BODY> -> "{"(<SYSA> "," <COMP> "," | <SYSA> "," <SPLTJ> "," )+ <SYSA> "}"

<SPLTJ> -> "SplitJoin" "{" ( <SCH> "," )* <SCH> "}"

<COMP> -> "Component" "{" "Name:" (<digit> | <letter>)+ "}"

<SYSA> -> "SysAssert" "{" "Name:" (<digit> | <letter>)+","
                          "Assert:" <predicate> "}"
```

Example

The following example demonstrates a textual system contract for the M1 example.

```
SystemContract = Contract {
    SysAssert {
                Name: START_SysAssert
                Assert: a1 != b1
    },
    SplitJoin {
        Schema{
            SysAssert {
                Name: RID_AC1A_SysAssert
                Assert: a1 != b1
            },
            Component {
                Name: AC1A
            },
            SysAssert {
                Name: RID_AC1A_End_SysAssert
                Assert: a = b1 && a1 != b1
            }
        },
        Schema{
            SysAssert {
                Name: RID_AC1B_SysAssert
                Assert: a1 != b1
            },
            Component {
                Name: AC1B
            },
            SysAssert {
                Name: RID_AC1B_End_SysAssert
                Assert: b = a1 && a1 != b1
            }
        }
    } (* IMPLICT JOIN *),
    SysAssert {
        Name: RID_AC2_SysAssert
        Assert: a = b1 && b = a1 && a1 != b1
    },
    Component {
        Name: AC2
    }
    SysAssert {
        Name: END_SysAssert
        Assert: a1 = a && b1 = b && a1 != b1
    },
}
```

Chapter 5

Verification Conditions and Independence Requirements

5.1 Component Contract Verification Condition

The *component contract verification condition* is used to verify that a component's action satisfies its corresponding contract. For a component `C`, system state `st`, and system `SYS`, the verification condition can be formally stated as

$$
\begin{array}{|l}
\hline
Pre_C\ (getLocalSt\ (TID_C)\ (st)) \\
tscs' = applyAction\ (SYS)\ (TID_C)\ (st) \\
\vdash Post_C\ (getLocalSt\ (TID_C)\ (st))\ (tscs') \\
\hline
\end{array}
$$

Note: `getLocalSt` is a function that returns the local state of a component given its ID and the system state, and `applyAction` is a function that applies the action associated with the component to the global state and returns the updated local state of the task.

This verification condition only exists within the Isabelle model and will not be present in the final HAMR implementation, as it serves as a stand-in for tools such as Logika[13;16] or Verus[17], which will verify that a component satisfies its contract.

5.2 System Contract Verification Condition

The *system contract verification conditions* verify that the assertions associated with places that are added to the schedule state, as a direct result of firing a transition or resetting the schedule, hold for the resulting state. Additionally, they establish the connection between the component contracts and system contracts by establishing strength requirements on the pre-assertions of component transitions. Finally, they establish that the initial state satisfies the assertion that should hold at the start of the schedule.

The following subsections will introduce the system contract verification conditions.

5.2.1 Initial State Verification Condition

The *initial state verification condition* requires that the initial system state must satisfy the START assertion (i.e., the assertion tied to the START place). This verification condition, while simple, is required for proving the soundness of the system and acts as the base case for the proof. Given the initial state of the system `SYS`, the verification condition can be stated formally as

$$\vdash START\_Assert\ (initState\ (SYS))$$

An example of this verification condition for the M1 example can be seen in Figure 5.1.

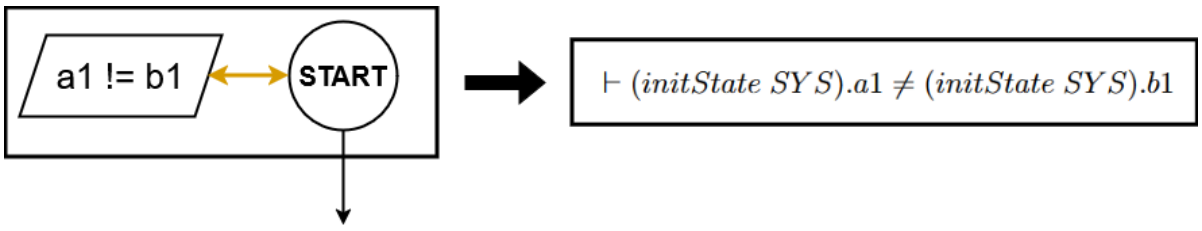

Figure 5.1: *M1 Initial State VC*

5.2.2 Pre-Assert Verification Condition

The *pre-assert verification condition* requires that, if a transition is a component transition and a state satisfies all pre-assertions for the transition, then the state should satisfy the precondition of the corresponding component. The purpose of this verification condition is to require that the pre-assertions of a component transition be strong enough to establish the precondition of the component and, thus, be able to connect the system assertions to component contracts. This connection, along with the component write frames, allows the system-level reasoning framework to use the component contract-defined behavior for verifying system assertions and reasoning about updates to the system state. For some system state `st`, component transition `T` with an associated component `C`, and a sets-and-maps representation of the system contract `sysSpec`, the verification condition can be formally stated as

$$\forall p \in In(T).\ (sysSpec(p))\ (st)$$
$$\vdash PRE_C\ (getLocalSt\ (TID_C)\ \ (st))$$

An example of this verification condition for the AC2 component of M1 can be seen in Figure 5.2.

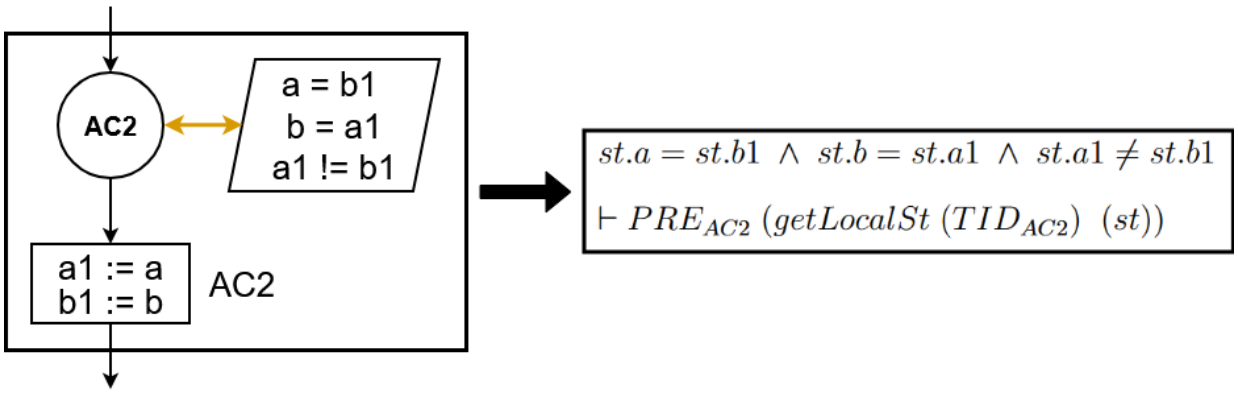

Figure 5.2: *M1 AC2 Pre-Assert VC*

5.2.3 Next-Assert Verification Condition

The *next-assert verification condition* requires that, if a pre-state satisfies the pre-assertions of a transition, then the post-state that results from firing the transition satisfies each individual post-assertion of the transition. The purpose of this verification condition is to verify that the pre-assertions and the definition of corresponding transition imply the post-assertions. There are two cases of this verification condition corresponding to the two types of transitions, as they differ in execution semantics.

The first case is when a transition is a control point transition, which only updates the schedule state and not the task or channel states. Therefore, if all of the pre-assertions hold for some pre-state, then the post-assertions should also hold for the post-state that results from firing the transition. Given a control point transition `T`, a system state `st`, an out-place of `T p'`, and a sets-and-maps representation of the system contract `sysSpec`, the verification condition can be formally stated as

$$(\forall p \in In(T).\ (sysSpec(p))\ (st))$$
$$\vdash (sysSpec(p'))\ st$$

An example of this verification condition for the JOIN transition of M1 can be seen in Figure 5.3.

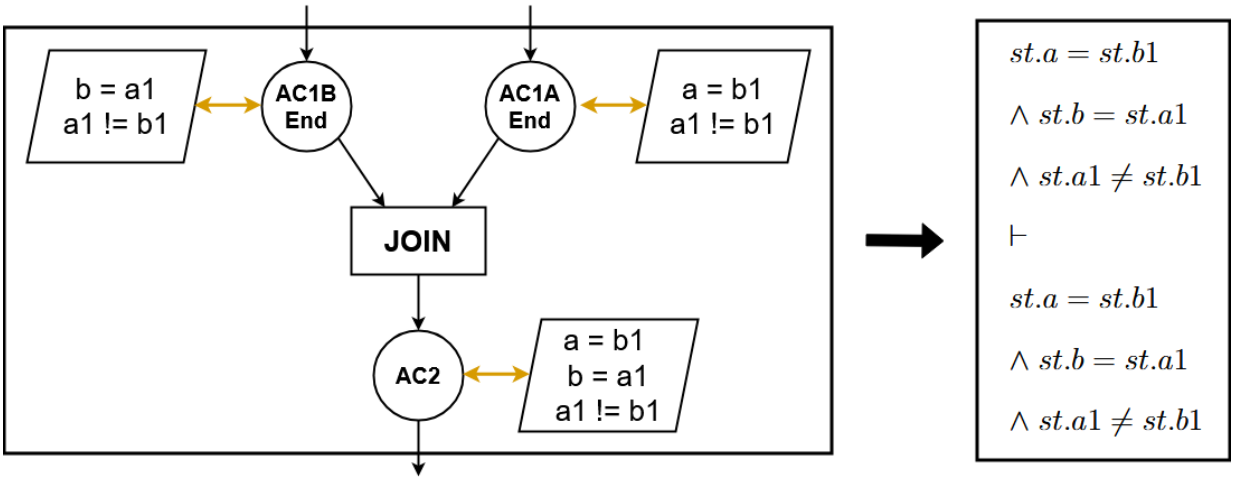

Figure 5.3: *M1 JOIN Next-Assert VC*

The second case is when a transition is a component transition. Because the corresponding component has an action that updates the system state, the verification condition needs predicates that express the execution of the action, in addition to the requirement that the pre-state satisfies the pre-assertions. To achieve this, the postcondition of the component and the local and global write frames are added as preconditions. Given a component transition `T` with an associated component `C`, a system pre-state `st1`, a system post-state `st2`, an out-place of `T p'`, and a sets-and-maps representation of the system contract `sysSpec`, the verification condition can be formally stated as

$$
\begin{aligned}
&(\forall p \in In(T).\ (sysSpec(p))\ (st1)) \\
&\wedge\ Post_C\ (getLocalSt\ (TID_C)\ (st1))\ (getLocalSt\ (TID_C)\ (st2)) \\
&\wedge\ LocalFrame_C\ (st1)\ (st2) \\
&\wedge\ GlobalFrame_C\ (st1)\ (st2) \\
&\vdash\ (sysSpec(p'))\ (st2)
\end{aligned}
$$

An example of this verification condition for the AC2 component of M1 can be seen in Figure 5.4.

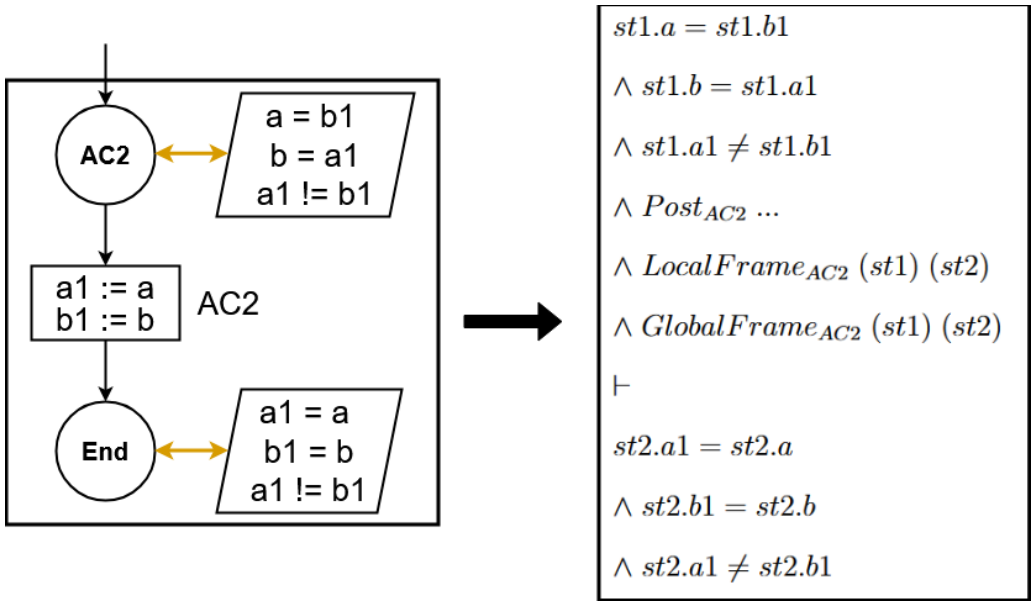

Figure 5.4: *M1 AC2 Next-Assert VC*

5.2.4 Post-Pre Verification Condition

The *post-pre verification condition* requires that the END assertion (i.e., the assertion tied to the END place) should imply the START assertion. This verification condition is meant to establish that the START assertion is a loop invariants of the schedule and therefore establish that the contract holds for all execution cycles. For a system state `st` where the schedule state is $\{|END|\}$, this verification condition can be stated formally as

$$END\_ASSERT\ (st)$$
$$\vdash START\_ASSERT\ (st)$$

An example of this verification condition for the M1 example can be seen in Figure 5.5.

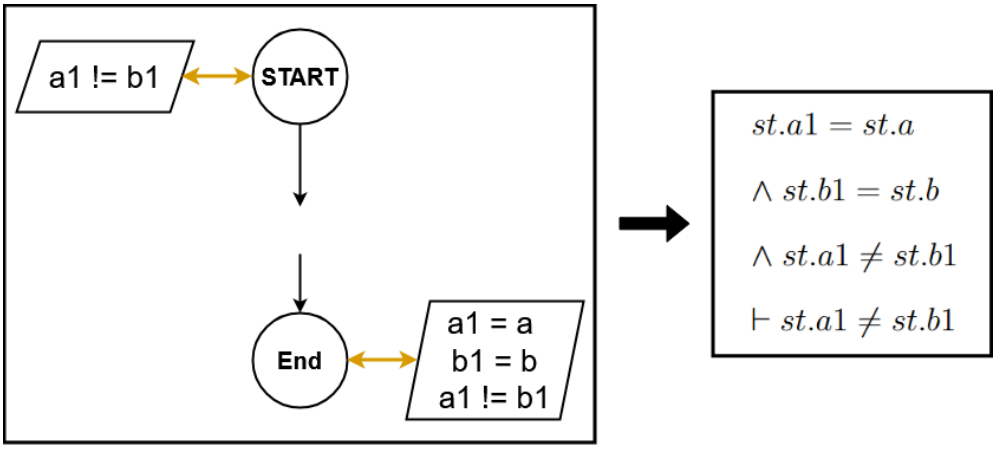

Figure 5.5: *M1 Post-Pre VC*

5.3 Independence

While the system contract verification conditions are able to verify that the assertions associated with places that are added to the schedule state, due to firing transitions or resetting the schedule, hold in the resulting state, the resulting schedule state may contain places that are not with associated the fired transition and were preserved from the previous schedule state. The assertions tied to these places still need to be satisfied by the system state, by the definition of the contract. Therefore, a formalism is necessary to ensure that the assertions still hold true under execution. To address this, we provide a definition of concurrence and independence to be able to prove the preservation of such assertions.

A pair of transitions is concurrent if and only if there exists a reachable schedule state where both transitions are enabled. Given a transition T_1 and a transition T_2, they are concurrent if and only if

$$\vdash \exists M.\ \{|START|\} \xrightarrow{*} M \ \wedge\ T_1 \subseteq M \ \wedge\ T_2 \subseteq M$$

We require that all pairs of transitions that satisfy the definition of concurrence are also independent, where a pair of transitions is independent if and only if they are non-blocking and they preserve the post-assertion of the other transition. These conditions enable reasoning about the preservation of assertions tied to transitions that are concurrent to the transition that was chosen to fire.

As an example of concurrence, if we were to analyze the set of reachable states in the M1 example, it could be seen that AC1A and AC1B are concurrent and, therefore, must be independent.

The following subsections will define each of the requirements for independence in detail and prove that AC1A and AC1B satisfy these requirements.

5.3.1 Non-Blocking Requirement

Without loss of generality, given a component transition T_1 and transition T_2, T_1 does not block T_2 if and only if, given a pre-state that satisfies the pre-assertions of both transitions, firing T_1 produces a post-state that satisfies the pre-assertion of T_2. This requirement establishes that, if two distinct transitions are enabled in a schedule state and one is chosen to fire, then the one that fires will not negate the pre-assertions of the other.

Given a system state `st`, a component transitions T_1, a transition T_2, and a sets-and-maps representation of the system contract `sysSpec`, non-blocking can be formally defined as

$$
\begin{array}{l}
(\forall p \in In(T_1).\ (sysSpec(p))\ (st)) \\
(\forall p \in In(T_2).\ (sysSpec(p))\ (st)) \\
\wedge\ st' = applyActionGlobal\ (SYS)\ ((activationMap\ (SYS))\ (In(T_1)))\ (st) \\
\vdash \forall p \in In(T_2).\ (sysSpec(p))\ (st')
\end{array}
$$

Note: `applyActionGlobal` is a function that, given a system, component ID, and system state, will apply the action of the component to the system state and return the new system state.

For the M1 example, to prove that AC1A and AC1B are non-blocking, the following conditions must be proven to be true.

$$
\begin{array}{l}
st.a1 \neq st.b1 \\
\wedge\ st' = applyActionGlobal\ (M1System)\ (TID_{AC1A})\ (st) \\
\vdash st'.a1 \neq st'.b1
\end{array}
$$

$$
\begin{array}{l}
st.a1 \neq st.b1 \\
\wedge\ st' = applyActionGlobal\ (M1System)\ (TID_{AC1B})\ (st) \\
\vdash st'.a1 \neq st'.b1
\end{array}
$$

5.3.2 Preservation of Post-Assertions

Without loss of generality, given a component transition T_1 and a transition T_2, T_1 preserves the post-assertions of T_2 if and only if, given a pre-state that satisfies the pre-assertions of T_1 and the post-assertions of T_2, firing T_1 produces a post-state that satisfies the post-assertions of T_2. This requirement establishes that if an assertion tied to a place in the schedule state is a post-assertion of a transition that is concurrent to a transition that is chosen to fire, then the assertion holds for the state that results from firing the chosen transition. Given a system state `st`, a component transition T_1, a transition T_2, and a sets-and-maps representation of the system contract `sysSpec`, preservation of post assertions can be formally defined as

$$
\begin{array}{l}
(\forall p \in In(T_1).\ (sysSpec(p))\ (st)) \\
(\forall p \in Out(T_2).\ (sysSpec(p))\ (st)) \\
\wedge\ st' = applyActionGlobal\ (SYS)\ ((activationMap\ SYS)\ (In(T_1)))\ (st) \\
\vdash \forall p \in Out(T_2).\ (sysSpec(p))\ \ (st')
\end{array}
$$

For the M1 example, to prove that AC1A and AC1B preserve the post-assertion of the other transition, the following conditions must be proven to be true.

$$
\begin{array}{l}
st.a1 \neq st.b1 \\
\wedge\ st.b = st.a1 \\
\wedge\ st' = applyActionGlobal\ (M1System)\ (TID_{AC1A})\ (st) \\
\vdash st'.b = st'.a1\ \wedge\ st'.a1 \neq st'.b1
\end{array}
$$

$$
\begin{array}{l}
st.a1 \neq st.b1 \\
\wedge\ st.a = st.b1 \\
\wedge\ st' = applyActionGlobal\ (M1System)\ (TID_{AC1B})\ (st) \\
\vdash st'.a = st'.b1\ \wedge\ st'.a1 \neq st'.b1
\end{array}
$$

5.4 Soundness

The system contract framework is considered sound if and only if any system state reachable from the initial system state over execution satisfies all assertions tied to its schedule state. Given a system `SYS`, system state `st`, and a sets-and-maps representation of the system contract `sysSpec`, the definition of soundness can be formally stated as

$$systemReach\ SYS\ st$$
$$\vdash \forall p \in (ready\ st).\ (sysSpec(p))\ (st)$$

Note: `systemReach` is the reflexive transitive closure of the definition of the execution step starting at the initial state.

Before presenting the proof of soundness for the system contract framework, the following assumptions must hold:

- The model is well-formed
- The system is well-formed
- The component contracts are well-formed
- The system contract is well-formed
- The system conforms to the component contracts
- The system conforms to the system contract
- The components of the system conform to the write frames
- `st1` is a well-formed system state
- `st1` satisfies all assertions tied to its schedule state
- `st2` is the system state resulting from taking a single execution step on `st1`

Under these assumptions, soundness is proven by induction on `systemReach`. The cases of the proof are as follows:

- **Base Case:** The initial system state satisfies the START assertion (this trivially holds by the Initial State verification condition).
- **Inductive Case:** If `st1` satisfies all assertions tied to its schedule state, then taking a single execution step produces `st2`, which satisfies all assertions tied to its schedule state.

  Given that the definition of an execution step has two cases, the inductive case can be further divided into the following cases:

  - **Component Transition Execution Case**
  - **Control Point Transition Execution Case**

Throughout the rest of this section, the proof for the two inductive cases will be provided on a high level, and the full proof can be found in *HAMRMicro05Spec.thy*[7] in the report's repository. Table 5.1 outlines the case and subcase structure for the inductive step of the soundness proof and can be used to navigate the rest of the proof.

Table 5.1: *Inductive step cases*

Case	**Associated VCs and Independence Requirements**	**Location**
Component Transition		Section 5.4.1
Choice Partition	Pre-Assert VC, Component Contract VC, Next-Assert VC	Section 5.4.1
Alt-Choices Partition	Non-Blocking	Section 5.4.1
JoinNotSat Partition	Preservation of Post-Assertions	Section 5.4.1
Control Point Transition		Section 5.4.2
Choice Partition is END	Post-Pre VC	Section 5.4.2
Choice Partition is Not END	Next-Assertion VC	Section 5.4.2
Not Choice Partition	N/A	Section 5.4.2

5.4.1 Component Transitions

Given all assumptions are true, to prove that `st2` satisfies all assertions tied to its schedule state over the firing of a component transition, the schedule state of `st1` is partitioned into three sets, being

- **Choice:** The set of in-places of the chosen enabled component transition
  - The chosen component transition will be denoted as **C**
- **Alt-Choices:** The set of places that are the in-places of any other enabled transition that is not **C**
- **JoinNotSat:** All places that do not belong to the other two sets
  - Due to the well-formedness requirements of the next-relation, all places in the set must be an in-place of an unsatisfied join transition.

For this inductive case, we show that all assertions tied to each partition of the schedule state of `st1` prove the system assertions tied to a corresponding partition of the schedule state of `st2`. A graphical representation of the proof is presented in Figure 5.6.

Note: It has been proven that the partitions of the schedule state of `st2` that are correlated to those of `st1`, together cover the schedule state of `st2`.

The following subsections will explain how the assertions tied to each partition of the schedule state of st1 verify the assertion of a corresponding partition of the schedule state of st2, given the contract and write frames of the component associated with **C**, the system assertions, and the system properties.

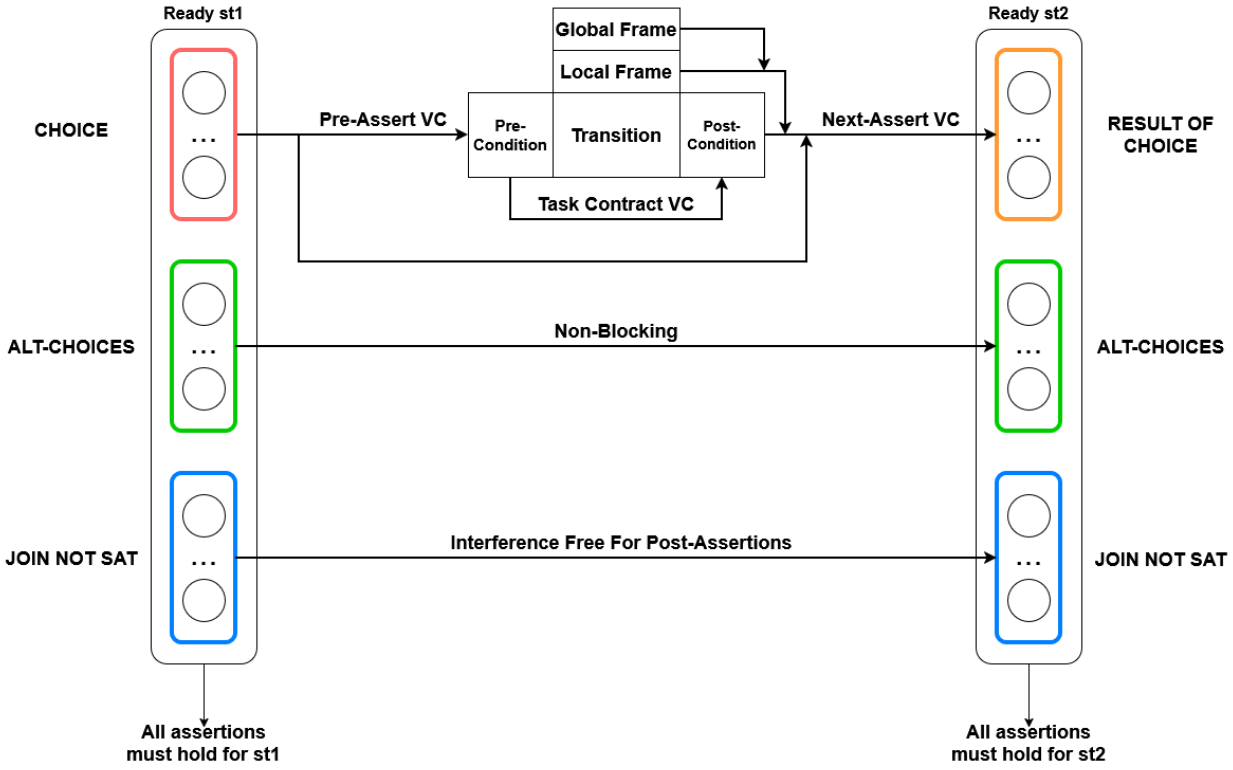

Figure 5.6: *Component case soundness diagram*

Choice

This partition is used to verify that because st1 satisfies the assertions tied to choice (i.e., the pre-assertion of **C**), st2 satisfies the post-assertions of **C**. The following is a high-level deconstruction of the proof given the assumptions:

st1 satisfies all assertions tied to its schedule state (i)
⇒ st1 satisfies all assertion tied to choice (pre-assertions of **C**) (ii)
⇒ st1 satisfies the precondition of the component tied to **C** because of the Pre-Assert Verification Condition (iii)
⇒ because st2 is the result of an execution step on st1, st2 satisfies the postcondition of the component tied to **C** because the Component Contract verification condition (iv)
From (i), (ii), and (iv), because the component conforms to its frame conditions, st2 satisfies the post assertions by the Next-Assert verification condition component transition case.

This section of the proof is represented by Figure 5.7.

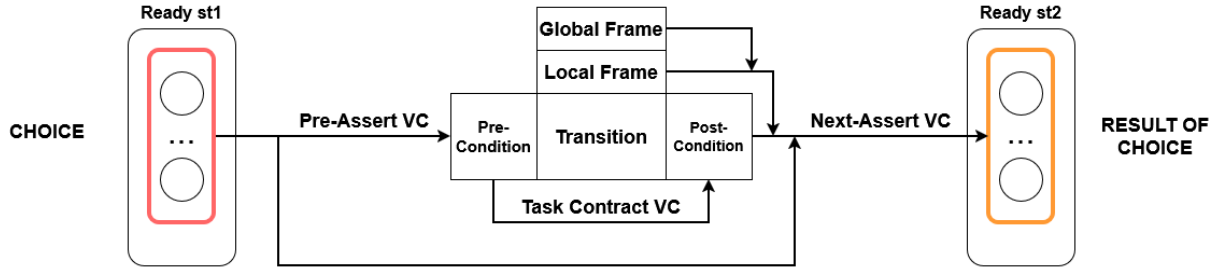

Figure 5.7: *Component case soundness diagram: Choice*

Alt-Choices

Because all transitions that are enabled in st1, that are not **C**, will still be enabled in st2, this partition verifies that all pre-assertions of all transitions that were enabled in the schedule state of st1, that are not **C**, hold for st2. The following is a high-level deconstruction of the proof given the assumptions:

st1 satisfies all assertions tied to its schedule state

⇒ st1 satisfies the pre-assertions of all alternate choices

⇒ because **C** and all alternate choices are enabled in st1 and the schedule state of st1 is reachable, **C** and all alternate choices are concurrent

⇒ choice and all alternate choices are independent

⇒ all pre-assertions for all alternate choices hold for st2 because of the non-blocking requirement for concurrent transitions.

This section of the proof is represented by Figure 5.8.

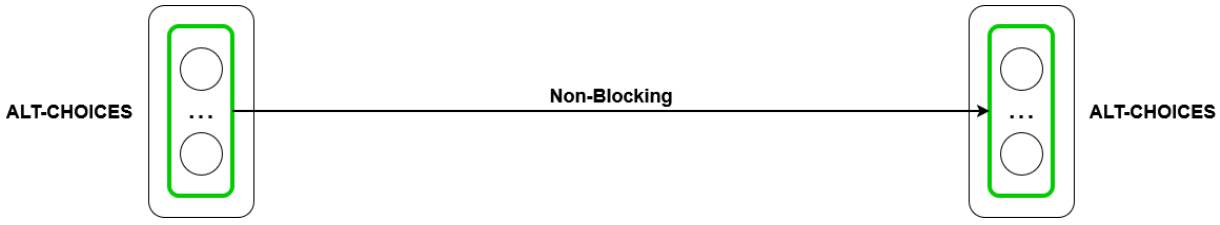

Figure 5.8: *Component case soundness diagram: Alt-Choices*

JoinNotSat

Because all places, not in Choice, will still be in the schedule state of st2, this partition is used to verify that all assertions tied to any remaining place in the schedule state of st1 hold for st2. For all places `p` in JoinNotSat, the following is a high-level deconstruction of the proof.

st1 satisfies all assertions tied to its schedule state

⇒ st1 satisfies the assert tied to `p` (i)

There exists a transition `T` whose set of out places is $\{|p|\}$, and `T` is concurrent to all enabled transitions in the schedule state of st1 by the wf_System_NextRel_JoinPrev and BackStepReachability well-formedness conditions.

⇒ `T` and **C** are concurrent

⇒ `T` and **C** are independent

⇒ Executing the component tied to **C** preserves the post-assertions of `T` (ii)

From (i) and (ii), the assertion tied to `p` must hold for st2 over the execution of the component tied to **C**

This section of the proof is represented by Figure 5.9.

Figure 5.9: *Component case soundness diagram: JoinNotSat*

5.4.2 Control Point Transitions

Given all assumptions are true, to prove that st2 satisfies all assertions tied to its schedule state over the firing of a control point transition, the schedule state of st1 is partitioned into two sets, being

- **Choice:** The set of in-places of the chosen enabled control point transition
  - The chosen control point transition will be denoted **C**
  - The proof for the Choice partition can be divided further into two cases: if Choice is $\{|END|\}$ and if it is not. This case split is done because the reset of the schedule state is not explicitly defined for the Petri net underlying the next-relation, and therefore, it has its own verification condition.
- **Not Choice:** The set of places that are not in Choice

For this inductive case, we show that all assertions tied to each partition of the schedule state of `st1` prove the system assertions of a corresponding partition of the schedule state of `st2`. A graphical representation of the proof is presented in Figure 5.10.

Note: It has been proven that the partitions of the schedule state of `st2` that are correlated to those of `st1`, together cover the schedule state of `st2`.

The following subsections will discuss how the assertions tied to each partition of the schedule state of `st1` verify the assertion of a corresponding partition of the schedule state of `st2`, given the system assertions and the system properties.

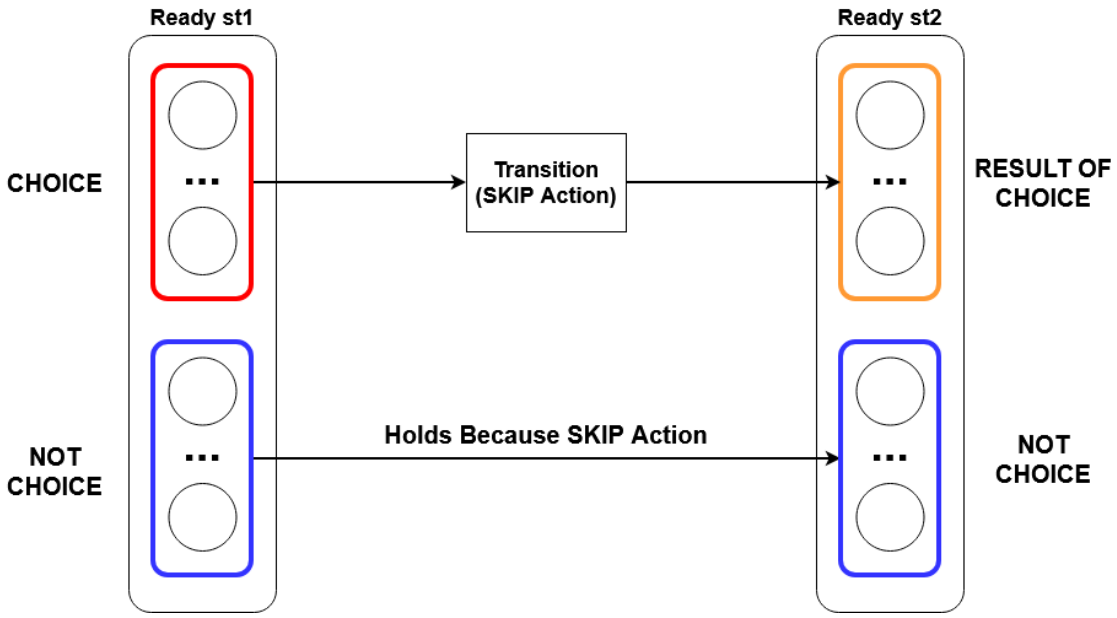

Figure 5.10: *Control point case soundness diagram*

Choice is END

This case of the Choice partition is used to verify that the assertion that can be made before any component is executed on a pass of the schedule is implied by the assertion that can be made after every component has been executed on a pass of the schedule. The following is a high-level deconstruction of the proof:

st1 satisfies all assertions tied to its schedule state
$\Rightarrow$ st1 satisfies the END assertion
$\Rightarrow$ st2 satisfies the START assertion due to the Post-Pre verification condition

This section of the proof is represented by Figure 5.11.

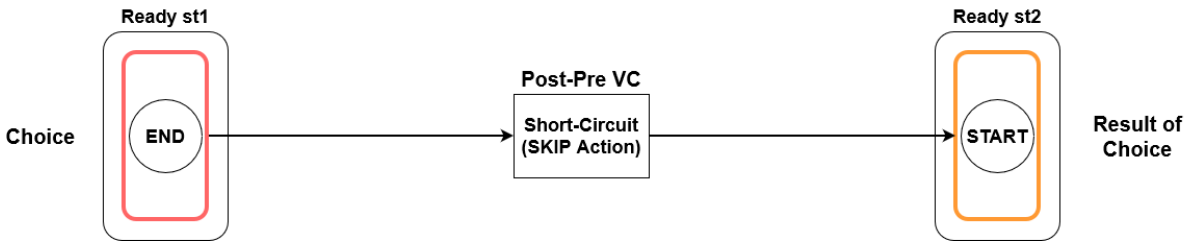

Figure 5.11: *Control point case soundness diagram: END Choice*

Choice is not END

This case of the Choice partition is used to verify that the assertions tied to Choice (pre-assertions of **C**) directly imply the post-assertions of **C** since the transition implicitly does not update the state. The following is a high-level deconstruction of the proof:

st1 satisfies all assertions tied to its schedule state
⇒ st1 satisfies the assertions tied to Choice
⇒ st2 satisfies the post-assertions of **C** due to the Next-Assert verification condition control point case

This section of the proof is represented by Figure 5.12.

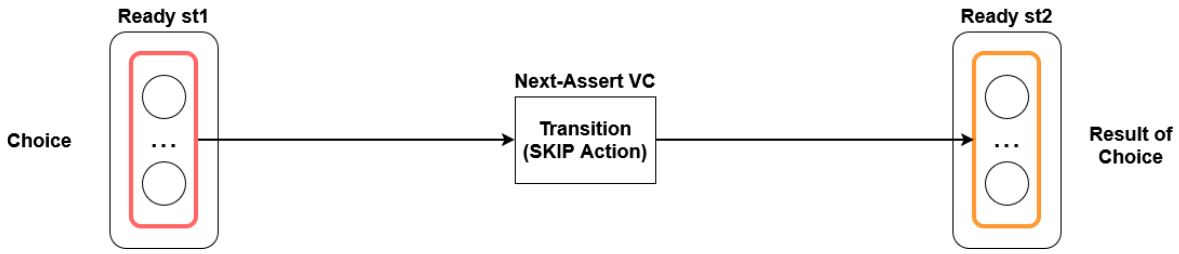

Figure 5.12: *Control point case soundness diagram: Not END Choice*

Not Choice

Because all places in the schedule state of st1, that are not in Choice, will still be in the schedule state of st2, this partition is used to verify that the assertion of any place in the schedule state of st1, that is not in Choice. is preserved over the firing of the transition. The following is a high-level deconstruction of the proof:

st1 satisfies all assertions tied to its schedule state
⇒ st1 satisfies the assertions of all places that are not in Choice
⇒ st2 satisfies the assertions of all places in the schedule state of st1, that are not in Choice, because the system state is not updated when a control point transition is executed

This section of the proof is represented by Figure 5.13.

Figure 5.13: *Control point case soundness diagram: Not Choice*

Chapter 6

Conclusions

This report has presented a sound theoretical foundation, using HAMR-Micro, for a system-level reasoning framework for AADL-aligned models with simple static scheduling for the HAMR toolkit. The approach enables the specification and verification of system-level contracts for systems with constraint-compliant schedules by leveraging the user-friendly formalism of a workflow net to represent the scheduling constraints. The HAMR-Micro formalization of the framework uses a workflow net representation of the constraints, component contracts, and a flexible non-deterministic execution semantics based on AADL-HSM to support the specification of system-level properties that must hold between the execution of explicitly constrained components in an execution cycle.

To ensure contract conformance, the framework emits verification conditions and independence requirements that can be checked using SMT-based tools. Furthermore, a high-level description of the soundness proof for HAMR-Micro formalization of the framework is presented to demonstrate that any possible execution of a system would result in a contract-conformant state.

Despite the contributions, there still exist several limitations of the framework. The first one is that, as noted in Section 4.2, the HAMR-Micro version of the framework is unable to derive the set of all valid schedules and therefore prove that any system that uses a constraint-compliant schedule will satisfy the system contract given a deterministic definition

of execution. However, checking that a static schedule conforms to the schedule constraints is a capability that can be implemented rather easily in HAMR, and due to several factors, it is reasonable to assume that all systems that use a constraint-compliant schedule will satisfy the contract in practice. Additionally, the breadth of expressible scheduling properties is intentionally restricted to simplify reasoning and align the semantics with pre-existing approaches for reasoning about parallel programs. Regardless, the expressible properties are sufficient for most systems developed using the HAMR toolkit.

6.1 Future Work

With the formalization of the framework, there exist several possible next steps.

HAMR toolkit implementation: The most immediate and important next step is prototyping the implementation of the framework in the full HAMR tool-chain. This will require several additions to the HAMR framework, including the implementation of the scheduling constraints and system assertion contracts into SysML v2, several preprocessing phases to check well-formedness, validation checks for proposed schedules, and the creation of several artifacts. The artifacts will extend several aspects of the HAMR tool-chain, such as the runtime monitoring, the property-based testing framework, and the verification framework. Further details of the implementation will be provided in future documentation of the framework that will include the algorithms to perform certain well-formedness checks, detailed descriptions of the artifacts, and illustrative examples of artifacts for several systems, such as the full Isolette.

Constraint refinement: The framework for the HAMR-Micro semantics could be extended to support the derivation of the set of all constraint-conformant linear schedules and the set of system assertions that should hold between the execution of each component in the schedule. This could be accomplished by establishing an inductive definition of a valid next-relation and adding the necessary definitions and lemmas to check that a provided next-relation satisfies the inductive definition. This extension would allow for all possible schedules to be generated systematically via refinement of the constraints, as the structure

of the next-relation would be more well-defined. Moreover, using a refinement approach would allow for further validation of the correctness of the framework by verifying that, if a system uses a constraint-conformant schedule, then it will satisfy the system-level contract. While prototyping the framework in the HAMR toolkit, the set of all valid schedules and the constraint-conformance of a proposed schedule will be derived by analyzing the set of all firing sequences across the reachability graph, and it will be assumed that any system that uses a schedule derived from the reachability graph conforms to the system contract.

Increased breadth of expressible properties: The framework for the HAMR-Micro semantics could be extended to support a wider breadth of scheduling properties. The current framework supports a very limited number of constraints compared to what sound WF-nets are capable of, such as the ability to express that the execution of a sequence of components in a schedule is able to be repeated. Even though the VCs would not change significantly, the proof of soundness would become much more involved as the progression of the schedule would be harder to generalize. Regardless, this effort would support a much more expressive framework that could work with a larger variety of systems.

Evaluate on more examples: In addition to the M1 example system that was used as the running example for this report, the framework has been applied to several other systems, such as the Isolette regulator module and a simple network guard. In each case, the framework was able to efficiently process and check the well-formedness and contract conformance of these systems. To further evaluate the framework, it should be applied to several larger and more complex systems to ensure the effectiveness and scalability of the approach.

Extension of the AADL-HSM: The formalization of the framework could be lifted to AADL-HSM, providing a direct link between the framework and the full semantics of HAMR and AADL. This would allow further validation of the correctness of the framework for the HAMR toolkit and enable the formalization of the framework to be used for future work with the AADL-HSM, such as connecting it to the Isabelle-based seL4 microkernel semantics.

Bibliography

[1] John Hatcliff, Jason Belt, Robby, and Todd Carpenter. Hamr: An aadl multi-platform code generation toolset. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, pages 274–295, Cham, 2021. Springer International Publishing. ISBN 978-3-030-89159-6. doi: 10.1007/978-3-030-89159-6_18. URL https://doi.org/10.1007/978-3-030-89159-6_18.

[2] John Backes, Darren Cofer, and Isaac Amundson. The assume guarantee reasoning environment with application to an unmanned helicopter. Technical report, Collins Aerospace, 2021. URL https://loonwerks.com/publications/backes2021techreport.html.

[3] Hamza Bourbouh, Pierre-Loïc Garoche, Thomas Loquen, Éric Noulard, and Claire Pagetti. Cocosim, a code generation framework for control/command applications an overview of cocosim for multi-periodic discrete simulink models. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, 1 2020. URL https://hal.archives-ouvertes.fr/hal-02441334.

[4] Airlines Electronic Engineering Committee et al. Avionics application software standard interface, arinc specification 653-1. *Avionics Application Software Standard Interface-ARINC Specification*, pages 653–1, 2003.

[5] Stefan Hallerstede and John Hatcliff. A mechanized semantics for component-based systems in the hamr aadl runtime. *Science of Computer Programming*, 245:103312, 2025. ISSN 0167-6423. doi: 10.1016/j.scico.2025.103312. URL https://www.sciencedirect.com/science/article/pii/S0167642325000516.

[6] Peter Feiler, David Gluch, and John Hudak. The architecture analysis & design language

(aadl): An introduction. Technical Report CMU/SEI-2006-TN-011, Feb 2006. URL https://doi.org/10.1184/R1/6584909.v1.

[7] Jacob Legg. Hamrmicro05, 2026. URL https://github.com/santoslab/hamr-system-reasoning-prototype/tree/main/IsabelleFormalization/HAMRMicro05.

[8] Carl Adam Petri. Kommunikation mit automaten. 1962. URL http://edoc.sub.uni-hamburg.de/informatik/volltexte/2011/160/.

[9] Carl Adam Petri. Communication with automata. 1966. URL http://edoc.sub.uni-hamburg.de/informatik/volltexte/2010/155/.

[10] W.M.P. van der Aalst. *Structural characterizations of sound workflow nets*. Computing science reports. Technische Universiteit Eindhoven, 1996. URL https://pure.tue.nl/ws/files/2454992/9712195.pdf.

[11] W.M.P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998. doi: 10.1142/S0218126698000043. URL https://doi.org/10.1142/S0218126698000043.

[12] Jorg Desel and Javier Esparza. *Free Choice Petri Nets*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1995.

[13] John Hatcliff, Danielle Stewart, Jason Belt, . Robby, and August Schwerdfeger. An aadl contract language supporting integrated model- and code-level verification. *Ada Lett.*, 42(2):45–54, April 2023. ISSN 1094-3641. doi: 10.1145/3591335.3591339. URL https://doi.org/10.1145/3591335.3591339.

[14] John Hatcliff, Jason Belt, Robby, Jacob Legg, Danielle Stewart, and Todd Carpenter. Automated property-based testing from aadl component contracts. In Alessandro Cimatti and Laura Titolo, editors, *Formal Methods for Industrial Critical Systems*, pages 131–150, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-43681-9. doi:

10.1007/978-3-031-43681-9_8. URL https://doi.org/10.1007/978-3-031-43681-9_8.

[15] John Hatcliff and Stefan Hallerstede. Hamrmicro04, 2025. URL https://github.com/santoslab/hamr-system-reasoning-prototype/tree/main/IsabelleFormalization/HAMRMicro04.

[16] Robby, John Hatcliff, and Jason Belt. Logika: The sireum verification framework. In Anne E. Haxthausen and Wendelin Serwe, editors, *Formal Methods for Industrial Critical Systems*, pages 97–116, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-68150-9. doi: 10.1007/s10009-025-00828-8. URL https://doi.org/10.1007/s10009-025-00828-8.

[17] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types (extended version), 2023. URL https://arxiv.org/abs/2303.05491.

Appendix A

WF Conditions For the Next-Relation

Table A.1: *Well-formedness conditions for the next-relation*

WF Condition	**Description**
wf_System_NextRel_1	The Petri net must have at least one transition
wf_System_NextRel_2	The set of all places in the Petri net must be a subset of the set of places in the model
wf_System_NextRel_3	The Petri net has a finite number of transitions
wf_System_NextRel_4	The Petri net has a finite number of out-place sets
wf_System_NextRel_5	All transitions must have a non-empty set of in-places
wf_System_NextRel_6	All transitions must have a non-empty set of out-places
mayHappenInParallelInd	If two transitions are concurrent, then they are independent
BackStepReachability	If a reachable marking M contains an in-place p of a join, then there exists a reachable marking M' where all enabled transitions, except the join, of M are enabled and the transition whose set of out places is $\{\|p\|\}$ is also enabled

Table A.2: *Well-formedness conditions for the next-relation cont.*

WF Condition	**Description**
wf_System_NextRel_DisjointDom	Petri net must be free choice
wf_System_NextRel_DomUnion	The union of the set of in-places for all transitions must be the set of all places except END
wf_System_NextRel_DisjointRan	Two unique transitions must have disjoint sets of out-places
wf_System_NextRel_SourceAndSink	START and END must be a source and sink for the Petri net respectively (WF-net requirement)
wf_System_NextRel_TaskTransition	A transition is a component transition if and only if it is a sequent transition
wf_System_NextRel_Split	If a transition has more than one out place then it must be a split transition
wf_System_NextRel_Join	If a transition has more than one in place then it must be a join transition
wf_System_NextRel_SplitNext	A split transition can only be succeeded by split and sequent transitions
wf_System_NextRel_JoinPrev	A join transition can only be preceded by join and sequent transitions
wf_System_NextRel_AllPossibleTransitions	A transition can only be a sequent, split, or join transition
wf_System_NextRel_NoReinsert	The set of in-places and the set of out-places for a transition must be disjoint
wf_System_NextRel_PlaceOnPath	All places appear on a path from START to END (WF-net requirement)
wf_System_NextRel_TransitionOnPath	All transitions appear on a path from START to END (WF-net requirement)
wf_System_NextRel_OptToComplete	Option to Complete (Soundness of WF-net)
wf_System_NextRel_ProperComplete	Proper Completion (Soundness of WF-net)
wf_System_NextRel_NoDeadTransitions	No Dead Transitions (Soundness of WF-net)
wf_System_NextRel_NoCycle	For all schedule states M that are reachable from $\{\|START\|\}$, M cannot contain the set of in-places and set of out-places for a transition (Acyclic)

Appendix B

M1 System Artifacts

Table B.1: *M1 system artifacts*

	Artifact Location
System Description	
Description	Section 2.2.1
AADL Model	Figure 2.2
Original Model Definition	Section 2.2.1
Original System Definition	Section 2.2.1
Original Initial State Definition	Section 2.2.1
Scheduling Constraints	
Scheduling Constraints	Section 3.1.4
WF-Net Representation of the Constraints	Figure 3.1
Next-Relation Representation of the Constraints	Section 3.1.4
All Constraint-Conformant Schedules	Section 3.1.4
Updated System Description	
Updated Model Definition	Section 3.2.1
Updated System Definition	Section 3.2.2
Updated Initial State Definition	Section 3.2.3
Component Write Frames	
AC2 Write Frames	Section 4.2

Table B.2: *M1 system artifacts cont.*

	Artifact Location
System Contract	
System Requirements	Section 4.3.2
Graphical Representation of the System Contract	Figure 4.3
Sets-and-Maps Representation of the System Contract	Section 4.4.2
Textual Representation of the System Contract	Section 4.4.3
Verification Condition Examples	
Initial State VC	Figure 5.1
Pre-Assert VC	Figure 5.2
Next-Assert VC Control Point Case [JOIN]	Figure 5.3
Next-Assert VC Component Case [AC2]	Figure 5.4
Post-Pre VC	Figure 5.5
Independence Requirements	
Non-Blocking	Section 5.3.1
Preservation of Post-Assertions	Section 5.3.2