UPPSALA
UNIVERSITET

IT 21 067
Examensarbete 30 hp
Juli 2021

Coroutines for Simics Device Modeling Language

Love Waern

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Coroutines for Simics Device Modeling Language

Love Waern

Coroutines have risen in popularity in modern programming primarily as an abstraction for non-blocking asynchronous logic. One particular domain where this is of interest is full-system hardware architecture simulation, as such systems heavily involve devices that communicate asynchronously. This thesis explores how coroutines may be designed for inclusion in the Simics Device Modeling Language -- used to develop device models for simulation with the full-system simulator Intel Simics. A conservative basic design has been developed, together with a number of experimental designs formed through iteration upon that basic design. Evaluation of these designs shows that the core elements of the basic design dramatically reduce boilerplate code compared to conventional approaches to asynchronous logic, but that the elements added by the experimental designs are rarely applicable without issue. The conclusion is that the approach is successful in developing an effective core design of coroutines, but further work and research is needed to determine what further extensions are necessary.

Handledare: Erik Carstensen
Ämnesgranskare: Konstantinos Sagonas
Examinator: Mats Daniels
IT 21 067
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Coroutines are a highly versatile programming abstraction which – in the most general sense – represent lines of execution that may choose to *suspend* their own execution, returning control to another part of the parent program. The execution of a suspended coroutine may then be resumed by the program at a later point [1].

Coroutines have risen in popularity as a means for idiomatically developing non-blocking *asynchronous procedures*; commonly presented through an interface called the `async`/`await` pattern.[1] Asynchronous procedures are procedures whose progression may involve waiting for external events to transpire – e.g. a message being received – but allow for other components of the program to execute while waiting. Coroutines are well suited for representing such procedures: a coroutine may suspend itself when an external event needs to occur – allowing the rest of the program to execute – and can later be resumed by the program once the event occurs. Although coroutines for the purposes of programming asynchronous procedures are most prominent within web development, they are also of interest in the domain of *full-system hardware simulation* – the simulation of multiple interconnected electronic devices forming a system architecture. Asynchronous communication between devices are extremely prevalent within such architectures. Coroutines have garnered interest for simplifying the development of *behavioral models* of such devices, which are necessary for simulation. A notable example of a full-system simulation framework which leverages coroutines is *SystemC* – a simulator and associated C++ library for developing device models to be used within that simulator. SystemC offers coroutines in the form of *processes* – lines of execution whose lifetimes span over the entire course of the simulation, and may suspend themselves in wait for a signal to be received, or for a period of simulation time to pass [7].

This thesis studies how coroutines may be designed for use in the development of device models for the full-system simulator *Intel Simics*. The primary means of developing device models for Simics is through the associated domain-specific language *Simics Device Modeling Language* (DML), which does not provide a coroutine abstraction, necessitating different means of representing asynchronous procedures. Specifically, the conventional approach is the use of *state machines*, where each state represents a unique waiting point of the asynchronous procedure; progression of the procedure is done through notifying the state machine of a particular event, causing it to act and transition accordingly. The implementation of such state machines involve large amounts of *boilerplate code*; because of this, it is highly desirable to extend DML with a more concise abstraction for representing

[1] Both Python and C++ explicitly present `async`/`await` as an interface for coroutines [9, 2]. Given the loose definition of coroutines, the `async`/`await` pattern arguably describes coroutines in most languages that adopt it – even if they are not called as such.

asynchronous logic, replacing the need for state machines.

The goal of this thesis is to research and develop viable designs of coroutines that DML may be extended with, in order to allow for the idiomatic development of event-driven asynchronous logic. The development of each design is done such that all identified Simics and DML-specific needs and issues are addressed.

In order to develop these designs, a primary *basic design* is developed and presented (Section 3), which emphasizes both *ease of use* and *ease and viability of implementation* – this ensures that the developed design is simple to use and understand, and minimizes the risk of severe issues that would complicate the design's implementation within the DML compiler and/or Simics. This design is then used as the basis of further iteration (Section 4); issues with the design are identified, discussed, and addressed through various means, developing new coroutine designs out of the basic one. Through this method, two main additional designs have been developed: the *bounded design*, which retains the same priorities and restrictions of the basic design, subsuming it, and the *unbounded design*, which is subject to a smaller subset of the restrictions that the basic and bounded designs are subject to, and de-emphasizes ease of implementation, performing only basic evaluations that implementation remains plausible.

The developed designs are evaluated by studying their impact on code when leveraged to rewrite a number of existing DML modules implementing asynchronous logic using state machines (Section 5). This is used to gauge the success of the developed designs, but also to compare differing aspects between the various design branches, and what impact these had. The results of the evaluation are then discussed and suggested changes to the designs are presented. Finally, conclusions are drawn (Section 6).

2 Background and Related Work

2.1 Coroutines

In the most general sense, coroutines are a generalization of subroutines, equipped with the ability to suspend their own execution – returning control to another part of the program, typically the caller of the coroutine. A suspended coroutine may later be resumed by another part of the program, at which point the coroutine continues execution at the point it previously suspended itself, up until the next suspension or the termination of the coroutine.

Due to the ability to be suspended and resumed, coroutines represent asynchronous lines of execution which are executed concurrently with the rest of the program. In this sense, coroutines are heavily related to *threads*; however, while context switching between threads are *preemptive* – and may occur at any point of a thread's execution – context switches between

coroutines are entirely *cooperative*; both suspension and resumption is done at explicitly defined points. Because of this, threads subsume coroutines in expressive power, as thread synchronization can be used in order to emulate the cooperative nature of coroutines.

The appeal of coroutines lies in their restricted nature compared to threads, as the lack of preemption allows for additional flexibility in implementation while also limiting the number of possible interactions between coroutines. In particular, compared to threads: [8]

- The resource footprint of coroutines and the speed of context switches can be greatly improved; as context switching is cooperative, there is no need for a mechanism to interrupt the execution of a coroutine, and no strict need for a central scheduler – although the latter may be desirable depending on the design and implementation.

- The execution of a program featuring coroutines can be made entirely deterministic, making it simpler to both write safe asynchronous code and to debug such code. In particular, the use of coroutines cannot give rise to *race conditions*, as the execution of a coroutine can only be interrupted when it allows itself to be.

  Although programs using coroutines may still feature bugs due to coroutines interacting incorrectly or in unexpected ways, the fact that such interactions may only occur at specific points, combined with the fact that execution is deterministic, makes it easier for the user to both avoid and resolve such issues.

Due to the above, coroutines have become increasingly popular as an alternative to threads for representing asynchronous logic which does not need to be preemptive in nature [5]. In particular, coroutines are suitable for *event-driven programming*.

2.2 Simics

Simics is a *full-system architecture simulator*; it is a framework for simulating entire systems of interacting electronic devices in enough level of detail to allow for software binaries developed for such an architecture to be tested via the simulator without the need to modify the software itself. Simics supports the simulation of both CPUs and peripheral devices, as well as memory systems, storage, interconnection buses and I/O [10].

Simics is designed to prioritize performance, testing and debugging capabilities, and collaboration features for the purposes of developing and testing system architectures. Simulation is done at a high level, aimed only to depict the behavior and interactions of entities within the system rather than to accurately replicate their internal workings. This allows users to

model architectures within Simics through high-level specifications, while also causing the simulation of such architectures to be extremely efficient.

Each entity in a Simics virtual environment is called a configuration object, and is either:

- An instance of a predefined component type which is given specialized treatment by Simics. This includes processors running specific instruction set architectures – which are simulated through virtualization.
- An instance of a user-defined *model* of any electronic device.

A device model is a high-level specification of the behavior of the device, as well as any associated resources and interfaces that are relevant for interacting with the device – such as what registers are associated with the device, and how these are memory-mapped.

Simics provides multiple means of developing device models: [4]

- Leveraging the Simics API – which exists for C, C++, and Python – in order to define the components of a device.
- Integrating SystemC device models (see Section 2.5)
- DML

The operation of a modeled device is completely driven by *events* propagated through the simulation engine; behavioral code of a device model may be triggered from one of the following:

- A synchronous invocation from another device via an established inter-device connection.
- Memory-mapped I/O; behavioral code may be associated with the contents of mapped register being modified.
- A timed event previously queued by the modeled device.

Any interaction that causes behavioral code of a device model to be executed is called a *device entry*.

A device model may feature code not associated with the device's operation during the simulation itself, but is instead directly invoked by the simulator for the purposes of initialization and configuration, checkpointing, or debugging.

2.2.1 Checkpointing

A key debugging feature of Simics is *checkpointing*, which is the ability to reify the state of the simulated system at a point in time, and permanently store it on the file system in a portable form. A checkpoint may be restored

in order to bring a simulated system to the exact state it was in when the checkpoint was created. Simics also allows for checkpoints to be continuously created during the execution of a simulation, which may then be leveraged in order to smoothly progress a simulation *backwards* in time – a feature called *reverse execution.*

In order to save a checkpoint, the state of the simulated system must be able to be *serialized* – able to represented in a compact form which is independent from the current state and architecture of the host system. When developing a device model, it is the developer's responsibility to identify all pieces of mutable state needed to faithfully recreate the state of device, and ensure that these are serialized and stored when a checkpoint is made, and restored when a checkpoint is resumed.

2.2.2 Checkpoint Compatibility

In addition to the requirement that all key state of a device must be serializable and stored in checkpoints, such serialization should ideally be performed in such a way to maximize maintainability as the model changes. This desirable quality of devices model is called *checkpoint compatibility*, and have three main criteria:

(a) All necessary state should be serialized in a checkpoint, and de/serialization should be deterministic.

(b) De/serialization is performed in such a way to make it resilient to changes in the model – ideally only causing issues if the model undergoes significant changes.

(c) Serialized state should be represented in an accessible, human-readable format, such that if de/serialization issues do occur from changes in the model, the user should be able to easily address these by modifying the stored serialized state.

2.3 DML

DML is a domain-specific language for developing device models to be simulated with Simics. It is an object-oriented language, where each device is represented through an object, which – as members – may feature pieces of mutable state, configurable parameters and attributes, subroutines (called methods), and subcomponents. Subcomponents, in turn, are objects that may have their own members. In DML terminology, objects which may be composed of other objects are specifically referred to as *compound objects*, while other declared entities (such as methods and mutable variables) are referred to as *non-compound objects.*

In contrast to typical general-purpose object-oriented languages – e.g. C++, C# and Java – objects in DML are statically declared rather than

dynamically created. In addition, any resources that may persist over the course of the simulation are also typically statically allocated. DML provides a single mechanism for dynamically allocating memory on the heap – roughly corresponding to `malloc` – and resources allocated this way have no intrinsic means of being serialized.

DML is syntactically similar to C, and indeed, shares many elements with C and offers limited interoperability. The DML compiler – *DMLC* – uses C as the target language, often generating code through simple transcription. In particular, DML's type system for run-time values is an extension upon C's type system.

DML has a static type system with relatively strong expressive power in comparison to C, but weaker than C++, C# and Java. A particularly notable metaprogramming feature that DML offers are *templates*, which allow for defining a block of *object statements* – such as method and member declarations – to be inserted into declared compound objects by *instantiating* the template. DML also provides features that allow a user to generically manipulate or traverse instances of a particular template – either globally throughout the entire DML program or locally within a particular scope. A notable shortcoming of DML templates is that they do not support associated type parameters (*generics*), limiting their expressive power.

The DML language has two main dialects: DML 1.2 and DML 1.4. DML 1.2 is an older version of the language still supported for backwards compatibility. This thesis concerns DML 1.4 as supported by Simics version 6.0.76.

This section will consist of a brief, non-exhaustive description of the most important elements of DML needed to understand presented code and discussion featured in this report.[2]

2.3.1 Overview

A complete DML program specifies exactly one device model, together with:

- Associated register banks, and how these may be memory mapped
- Specifications of connections to other devices that the device expects to have access to, and thus may make use of.
- Specifications of connections that other devices may establish to the device, and how messages sent through those connections are handled by the device.
- Specification of meta-attributes that the configuration environment may access for the purposes of configuring the device before the simulation, gain introspection into the device, or to *checkpoint* the device state.

[2]For a more comprehensive description of DML, see the DML 1.4 Reference Manual [6].

- The name and description of the device, and other static meta-information

These are the crucial properties of the device model that must be made visible to Simics, and each of these have specialized language features in order to declare them.

Beyond these, the DML language is host to a number of features that exist only to improve the expressive power of language and simplify development; for instance, templates are a powerful metaprogramming tool that allows for code reduction and reuse, as well as a means of building abstractions. DML also features a basic exception mechanism for error handling; built-in syntax for bit-slicing; and built-in statements for logging and assertions. Furthermore, DML also offers *event objects* that allow developers to take advantage of the support Simics provides for posting *timed events* to be triggered at a later specified point during the simulation – at which point a specified callback is executed.

Analogously to C, a DML program may span multiple modules; a DML module may import another, effectively inserting the contents of the imported module into the importing module. Every DML program must feature exactly one *device* object, which serves as the declaration of the device that the program corresponds to. Unlike other compound objects within DML, device objects are not declared with an explicit body – instead, all other declared top-level objects, parameters, methods, and variables are considered members of the device object. A common pattern within DML is to separate a program into several modules, which are then imported by a central module which declares the device; for example:

```
// Language version specification
dml 1.4;

// Device object declaration
device example_device;

// Other modules containing the core logic.
import "banks.dml"
import "connections.dml"
import "communication_logic.dml"

// Parameter declarations that specify the name and description of
↪  the device.
// These are required.
param name = "Example device"
param desc = "A device only meant to demonstrate DML code"
```

2.3.2 Methods and Mutable Variables

Methods are the DML representation of subroutines. They may be declared as members of any compound object or template. Any method may have

multiple input parameters, specified similarly as C functions. Unlike C, DML methods may have multiple return values, and the lack of a return value is indicated through an empty list of return values rather than `void`. The following is demonstrates a method declaration with no input or output parameters:

```
method noop() -> () {
  return;
}
```

Alternatively:

```
method noop() {
  return;
}
```

The following demonstrates a method declaration with multiple input and output parameters:

```
method div_mod(uint64 dividend, uint64 divisor)
    -> (uint64, uint64) {
  local uint64 quot = dividend / divisor;
  local uint64 rem  = dividend % divisor;
  return (quot, rem);
}
```

This also demonstrates how local, stack-allocated variables within methods may be declared; through the `local` keyword. This is analogous to C's `auto` variable kind – but unlike C, the keyword must be explicitly given.

DML features two other variable kinds: `session` and `saved`. Unlike `local` variables, `session` and `saved` variables may also be declared as members of any compound object within the DML program, and can only be initialized with constant expressions.

`session` variables represent statically allocated variables, and act as the DML equivalent of `static` variables in C. The value of a `session` variable is preserved for duration of the current *simulation session*, but *are not* automatically serialized and restored during checkpointing. This means that it is the model developer's responsibility to manually serialize and restore any `session` variables upon saving or restoring a checkpoint.

`saved` variables behave exactly like `session` variables, except the value of `saved` variables *are* serialized and restored during checkpointing. Because of this, a `saved` variable must be of a type that DML knows how to serialize. Most built-in non-pointer C types are serializable, and any `struct`[3] that consists solely of serializable types are also considered serializable. Pointers are *never* considered serializable.

[3] As well as `layout`, a similar language feature in DML. DML does not support union types.

Methods have access to a basic exception-handling mechanism through the `throw` statement, which raises an exception without associated data. Such exceptions may be caught via the `try { ... } except { ... }` statement. If a method may throw an uncaught exception, that method must be declared `throws`; for example:

```
method demand(bool condition) throws {
  if (!condition) {
    throw;
  }
}
```

2.3.3 Templates and Parameters

A *template* specifies a block of code that may be inserted into compound objects. Templates may only be declared at the top-level, which is done as follows:

```
template name { body }
```

where *name* is the name of the template, and *body* is a set of object statements.

A template may be instantiated through the `is` object statement, which can be used within either objects, or within templates. For example:

```
bank regs {
  // Instantiate a single template: templateA
  is templateA;

  // Instantiate multiple templates: templateB and templateC
  is (templateB, templateB)

  register reg size 1 @0x0;
}
```

The `is` object statement causes the body of the specified templates to be injected into the compound object or template in which the statement was used.

`is` can also be used in a more idiomatic fashion together with the declaration of an compound object or template as follows:

```
// Instantiate templates templateA,
// templateB, and templateC
bank regs is (templateA, templateB, templateC) {
  register reg size 1 @0x0;
}
```

A language feature closely related to templates are *parameters*. A parameter is a *compile-time constant expression* that is a member of a particular

compound object or template.[4] Parameters may optionally be declared without an accompanying definition – which will result in a compile-time error if not overridden – or with a *default*, overridable definition. Parameters declared this way can be overridden by any later declaration of the same parameter. This can be leveraged by templates in order to declare a parameter that the template may make use of, while requiring any instance of the template to provide a definition for the parameter (or allow instances to override the default definition of that parameter).

Parameters are declared as follows:

- Without definition:

  `param` *`name`* `;`

- With overridable definition:

  `param` *`name`* `default` *`value`* `;`

- With unoverridable definition:

  `param` *`name`* `=` *`value`* `;`

Much of the DML infrastructure – as well as DML's built-in features – rely heavily on templates. Due to the importance of templates, DML features a wide variety of features to generically manipulate and reference template instances – both at compile time and at run time. These will not be detailed, as they are not relevant for the contents of this report.

2.3.4 Interaction with Simics – Attributes and Initialization

Device models have limited means for interacting with the Simics simulator itself – namely, initialization methods and attributes.

Initialization methods are used in order to initialize the state of a device model once an instance of the model is first created within a simulation. There are two initialization phases: initialization, and finalization – also called post-initialization. By instantiating the `init` or `post_init` templates and defining the corresponding methods of the same name, a DML program may perform arbitrary initialization for any declared compound object.

Device models may expose a number of *attributes* associated with the device, which are exposed to the Simics simulator, but not to other configuration objects within the simulation. Such attributes may be used in order to gain information about the properties and state of a device during the course of the simulation, or may be used to configure the device – either during initialization, or dynamically in the middle of a simulation.

[4] Parameters declared at the top level are members of the device object itself.

Attributes are the underlying mechanism through which *checkpointing* is done – the current value of every attribute of a device is fetched and permanently stored upon saving a checkpoint, and upon restoring a checkpoint, every attribute becomes set with the corresponding value that was fetched during the checkpoint.

Saved variables, event objects, register objects, and connect objects implicitly create attributes to allow Simics to configure and checkpoint these. In addition, device models may explicitly create arbitrary attributes through *attribute objects*, but these will not be detailed within this report.

2.3.5 Register Banks – Memory-Mapped IO

DML allows for specifying memory-mapped banks of device registers of the modeled device. Every such bank has an internal memory space, and each register is mapped to specific addresses within the space of its containing bank.

A Simics environment can be configured to map the memory space of each bank within a device to software-accessible memory regions – allowing for memory-mapped I/O with the device.

The `bank` compound object type is used to declare a bank of device registers, and the `register` compound object type is used to declare a register within a bank. DML also offers the `group` compound object type, which may be used to declare a specific group of registers within a bank.

Bank objects may only be declared at the top level – thus, as part of the device of the DML program – while groups and registers may only be declared within bank objects and (other) group objects.

In order for a register to be mapped into the internal address space of the bank containing it, its *size* and *offset* relative to beginning of the address space must be specified. The DML compiler statically checks that registers declared inside of a bank do not overlap with one another.

The following DML code declares a single bank `example_bank`, and a single register within it `example_reg`, which is declared with a size of 8 bytes and address offset `0x01000`. This causes it to be mapped to the address range `0x0100` through `0x0107` within the address space of the bank.

```
bank example_bank {
  register example_reg size 8 @ 0x0100;
}
```

Register objects have an associated storage location for a `uint64` to represent the current value of the register. This is automatically checkpointed, and may be accessed within the device through the `set` and `get` methods provided as part of the register object type.

Register banks of a device may be memory mapped by the Simics simulator to an emulated memory region, making the registers of that bank

accessible to other devices and simulated CPUs. Any read or write to a memory-mapped register is based upon calls to respective `read` and `write` methods of the register object. By default, such calls simply read `set` and `get`, but by having the register instantiate the `read` and `write` templates, the developer may override that behavior.

2.3.6 Device Interfaces – Inter-Device Communication

A Simics configuration may establish unidirectional connections from instantiated devices to other configuration objects – which may include other devices. This can be used for communication between a device and other configuration objects.

A device model may specify a number of connections and associated interfaces that it may make use of to access other configuration objects, as well as a number of interfaces that the device itself implements, allowing it to be accessed by other device configurations through those interfaces.

A Simics interface is a collection of method signatures – any device which connects to another configuration object through an interface communicates with it through the use of the associated methods, and any device which implements an interface must provide definitions for each method of the interface. Interfaces themselves are defined through an associated *interface type*, which declares the name of the interface and its associated methods. How interface types may be declared is not relevant for the scope of the report.

A device model may declare a connection which it may use to access other configuration objects through a `connect` object. Every `connect` object may contain a number of `interface` objects, which are used to declare the interfaces associated with the connection that the model may make use of. Unlike other objects, the identifier given when declaring an interface object is not only used as the name of the object, but also to indicate the associated *interface type*. For example, the following declaration:

```
connect i2c_link {
  interface i2c_master_v2;
  interface i2c_slave_v2;
}
```

Defines a connection named `i2c_link`, with two interface subobjects named `i2c_master_v2` and `i2c_slave_v2`. This causes the DML compiler to locate the definitions of the corresponding interface types and insert the methods associated with them into the respective interface objects.

The DML model may not specify which specific device to connect to; instead, the Simics configuration establishes the connection from that device to another which implements the interface types.

A device model may define an implementation for a specific interface type via the `implement` object type. Just like for interface objects, the identifier

for an `implement` object specifies the interface type, and any `implement` object must define and implement the methods required by the specified interface type.

Implements can be declared top-level, or as part of a `port` object. Any connection to the device from another can be made either to the device itself – in which case the top-level interfaces implemented are accessible by the connecting object – or to one of its ports – in which case the interfaces implemented within that port are accessible to the connecting object. This allows a device model to expose different sets of interfaces to different configuration objects.

2.3.7 Timed Events

Simics allows devices to post *events*, to be triggered at a later point during the simulation. Such events are managed by Simics through *event queues*, which, in turn, are tied to a simulated CPU. This allows for configuring posted events to be triggered after a certain amount of simulated CPU *time*, *cycles*, or *steps*.

In DML, there are two means of posting events to Simics: the `after` statement, and `event` compound objects. For simplicity, only the `after` statement will be described in this section.

The `after` statement is meant for simple in-line event posting, and is used as follows:

```
after scalar unit: callback();
```

where *unit* must an identifier for a *unit of time*, *scalar* must be a valid scalar for that unit, and *callback* is an identifier for a zero-parameter method. Executing an `after` statement causes an event to be posted to the event queue to be triggered after the time specified; at which point *callback* is called.

As of Simics 6.0.76, the only supported time unit is CPU time in seconds, whose corresponding identifier is `s` [6]. For example, the following is a valid `after` statement:

```
after 1.2 s: foo();
```

The `after` statement has a number of shortcomings, which are addressed by the use of the more complicated `event` object type:

- The `after` statement doesn't permit the use of CPU cycles or CPU steps as units of time, even though these are supported by Simics.
- Only zero-parameter methods are supported as the *callback* – thus, no data can be associated with the posted event but the callback itself.

  This is because if parameters were supported, then they would need to be preserved until the event is triggered, and the callback executed. This

would be an issue as posted events need to be serializable – a checkpoint may occur between any event being posted, and it being triggered. As such, any provided parameters would need to be serializable, which is not possible for any arbitrary types.

- Once an event is posted through an `after`, there is no means for the device to manage the posted event – in particular, it's not possible to cancel it. This is problematic if the posted event becomes invalid due to changes in the state of the device. For example, a reset of the device may occur, and consequently, the device is placed in a state where it does not expect the callback to be executed.

2.3.8 Reset Signals

DML provides built-in support for various forms of reset signals and associated handling. There are three kinds of built-in reset signal types: *power-on reset* – representing reset from power loss, *hard reset* – typically representing circuit-level reset signals, and *soft reset* – typically representing software reset signals. Logic for handling a specific reset kind can be provided for a compound object by instantiating `power_on_reset`, `hard_reset`, or `soft_reset`, respectively, and defining the corresponding method of the same name. In addition, the `poreset`, `hreset`, and `sreset` templates can be instantiated to declare a port of the device through which the corresponding reset signal can be sent.

Register objects have built-in reset handling logic, which resets their contents to their initial values. This may be overridden by instantiating the relevant reset template as above; or disabled by instantiating the `sticky` or `no_reset` templates, the former of which disables the built-in reset handling on soft resets, while the latter disables it entirely.

2.3.9 Logging

DML provides a native statement for *logging messages*. This leverages the Simics API to allow a Simics user to selectively suppress logging emitted. A log statement has four main parameters:

`log` *log-type* `,` *log-level* `,` *log-groups* `:` *format-string* `,` e_1 `, ...,` e_N `;`

- *log-type*; the general category of the message. This must be one of the following:
  - `info` – miscellaneous messages giving introspection into the operation of the device.
  - `error` – internal logic error, indicating a bug inside the device model itself.
  - `spec_viol` – external logic error; the device is interacted with in a forbidden way.

  - `unimpl` – indicates a particular feature of the device is not implemented in the model.
  - `critical` – indicates a miscellaneous high-impact – typically unexpected – event which is not a true error.

- *log-level* and *log-groups*; used to restrict the scope of logging messages emitted to Simics: a user may restrict to only receive messages from a specified set of log groups, at or under a certain *log level*. Both of these parameters are optional.

  log-level must be an integer between 1 and 4, where 1 represents highest severity.

  log-groups specifies one or more previously declared log groups. A single log group can be specified by name; multiple are specified through bitwise-or syntax, e.g.:

  ```
  log info, 4, (loggroupA | loggroupB): "A message"
  ```

  A log group can be declared at top-level using the `loggroup` object statement:

  ```
  loggroup loggroupA;
  loggroup loggroupB;
  ```

- *format-string* and associated parameters: the message to be logged. This is a formatted string, analogous to that of C's `printf()` function.

2.4 Asynchronous Logic in Device Models

It often becomes necessary to model asynchronous processes that a device may perform over the course of the simulation – in particular, inter-device asynchronous communication such as through the I2C protocol.

Such asynchronous processes are driven by device entries – however, DML provides no native support for writing asynchronous logic spanning over multiple entries; it only allows specifying handlers per possible entry. This makes it necessary to leverage state machines in order to program asynchronous logic, which has historically proven to be an extremely boilerplate-heavy solution.

2.4.1 Example

In order to demonstrate the boilerplate intensive nature of asynchronous logic in DML, an example will be provided of a task involving asynchronous logic, together with a solution to that task leveraging state machines.

The task consists of implementing a model for a master device which communicates with two slaves via the I2C protocol; one slave *reads* data from

the master, and the other *writes* data to the master. Given the following definitions:

```
param Reader_Address = 0x0;
param Writer_Address = 0x1;


// Connection to slave
connect i2c_link {
  /*
  Offers:

  method start(uint8 target_address) -> ()
  method stop() -> ()
  method write(uint8 packet) -> ()
  */
  interface i2c_slave_v2;
}

// Handle slave responses
interface i2c_in {
  implement i2c_master_v2 {
    method acknowledge(i2c_ack_t ack_value) {
      // STUB
    }


    method read_response(uint8 response) {
      // STUB
    }
  }
}

bank control {
  register busy size 0 @0x0 is read_only {
    param init_val = 0;
  }
}

// should be called once first ack from reader slave is received
method initialize_writing() { ... }
// should be called once first ack from writer slave is received
method initialize_reading() { ... }

method make_packet() -> (uint8 packet) { ... }
method process_read(uint8 packet) -> () { ... }
method is_writing_done() -> (bool) { ... }
method is_reading_done() -> (bool) { ... }
```

The task is to implement the following:

- An `activate()` method which sets the `busy` register and initiates a write transaction to the reader.
- Once the write transaction is completed, initiate a read transaction from the writer.
- Once the read transaction has completed, clear the `busy` register.
- Leverage `i2c_link` and `i2c_in` in order to send messages to the slave, and handle responses from the salve, respectively (`acknowledge()` and `read_response()` must be implemented).

Using a state machine, one possible solution would be as follows:

```
param S_idle                       = 0;
param S_wait_for_reader_first_ack = 1;
param S_wait_for_writer_first_ack = 1;
param S_wait_for_write_ack        = 2;
param S_wait_for_read_response    = 4;

saved uint8 curr_state = S_idle;

method perform_read_request() {
  if (!is_reading_finished()) {
    i2c_link.read();
  } else {
    i2c_link.stop();
    curr_state = S_idle;
    control.busy.set(0);
  }
}

port i2c_in {
  implement i2c_master_v2 {
    method acknowledge(i2c_ack_t ack_value) {
      if (ack_value == I2C_ACK) {
        switch (curr_state) {
          case S_wait_for_reader_first_ack:
            initialize_writing();
            curr_state = S_wait_for_write_ack;
            // Intentional fall-through
          case S_wait_for_write_ack:
            if (!is_writing_done()) {
              local uint8 packet = prepare_write();
              i2c_link.write(packet);
            } else {
              i2c_link.stop();
              curr_state = S_wait_for_writer_first_ack;
              i2c_link.start(Reader_Address);
            }
```

```
            break;
          case S_wait_for_writer_first_ack:
            initialize_reading();
            curr_state = S_wait_for_read_response;
            perform_read_request();
            break;
          default:
            log error: "Illegal state";
            break
        }
      } else {
        log error: "no_ack received";
      }
    }

    method read_response(uint8 response) {
      if (curr_state == S_wait_for_read_response) {
        process_read(response);
        perform_read_request();
      } else {
        log error: "Illegal state";
      }
    }
  }
}

method activate() {
  control.busy.set(1);
  curr_state = S_wait_for_reader_first_ack;
  i2c_link.start(Reader_address);
}
```

In contrast, Appendix A demonstrates how the same problem may be solved using the basic coroutine design described in Section 3.

2.5 SystemC

SystemC is a C++ library forming an embedded domain-specific language for the modeling and subsequent simulation of multiple interacting entities, primarily used to model and simulate system architectures [3]. As such, its domain overlaps that of Simics – SystemC is used for system design and verification, and enables the development of embedded application and system software without the need for a silicon prototype or the register-transfer level design of the system at hand. Like Simics, SystemC device models are high-level behavioral specifications which are event driven in nature, in order to simplify system model development and allow for high-performance simulation.

Two notable differences with SystemC compared to Simics is the native

support for coroutines in the form of SystemC *thread processes*, and the lack of native support for checkpointing.

2.5.1 Coroutine Support

SystemC offers coroutines in the form of *thread processes*, which are *statically declared* behavioral components of a SystemC module. Any thread process is declared by specifying an associated C++ function which serves as the program executed by the thread process. Such functions may make use of the `wait()` function, which causes the calling thread process to suspend itself [7].

There are two kinds of thread processes: (normal) threads, and clocked threads, declared through `SC_THREAD` and `SC_CTHREAD` respectively:

- An `SC_THREAD` process may optionally be declared together with an associated *sensitivity list* of signals.
- An `SC_CTHREAD` process must be declared by specifying a clock signal it is dependent on, as well as if it should react to the positive edge or negative edge of that clock.

The `wait()` function can be used inside the body of any arbitrary C++ function, however, its behavior is only defined if executed by a thread process as part of executing the associated function of the thread process – either directly within the thread function, or indirectly within another function that the thread function calls.

The `wait()` function is overloaded, with differing behavior dependent on its arguments. The thread process kind specifically affects the behavior of `wait()` when called with no arguments:

- If the calling process is of kind `SC_THREAD`, then parameterless `wait()` will cause the process to be suspended until the a signal within its sensitivity list is changed.
- If the calling process is of kind `SC_CTHREAD`, then parameterless `wait()` execution will be suspended until the next specified edge occurs of the specified clock.

Thread processes may – instead of relying on their sensitivity list – call `wait()` upon one or more explicitly referenced *event objects*. Every event object has an associated `notify()` method, which causes all thread processes suspended upon the event to be resumed once the current thread process has suspended or terminated.

`wait()` supports waiting on multiple events simultaneously, with the option to either resume execution when one of the events are notified, or when all of them have been. This can be optionally combined with a timeout.

2.5.2 Simics Integration

Simics provides limited support for using SystemC modules in order to represent device models which may be simulated within Simics. Simics introduces the additional need for device modules to support checkpointing, requiring the modification of existing SystemC modules to provide de/serialize procedures which Simics may leverage. However, such de/serialization procedures are often not trivial – in particular, thread processes are problematic to serialize.

The SystemC standard doesn't specify nor has any recommendations on how thread processes should be implemented – however, as arbitrary C++ functions are allowed as the associated function of a thread process, it becomes necessary in practice to have separate execution contexts for each thread process – i.e. once a thread process becomes suspended, it's necessary to preserve the program counter, current stack and stack pointer, and register contents in order to be able to resume it later. Arbitrary execution contexts can't be serialized in general, and so Simics does not provide any native checkpointing of thread processes.

In order to allow modules containing thread processes to be serialized when integrated into Simics, it's the model developer's responsibility to keep track of the current position and surrounding context of a thread process and represent it in a serializable form during checkpointing; and then, upon resuming a checkpoint, restart each thread process and manually move execution within each process to the right point. This requires maintaining a state machine for each thread process, where the accompanying function is written such that each `wait()` is associated with a different state, and when resuming a checkpoint, the function will inspect the serialized state and jump execution to the corresponding `wait()` [11]. This leads to the same form of difficult, boilerplate-heavy code as the state machine equivalent solutions of the form of problems coroutines are intended to solve – thus eliminating the very reason SystemC thread processes are useful in the first place.

2.6 Concurrency Abstractions in Ada

Ada is a programming language designed for highly reliable and maintainable software systems [12]. Ada is notable in that it shares a property in common with DML – almost all components of an Ada program are statically declared. This includes Ada's core abstraction for concurrency – *tasks*. Ada tasks are not coroutines – as they are preemptively scheduled – but their design serves as a valuable reference due to their similarities in use.

Task are statically declared lines of execution that begin to run immediately upon the start of the program, and continue execution indefinitely over the program's course. Communication between tasks is done through synchronous *message passing* through *entry calls*. A task may declare a num-

ber of *entry points* – named sets of associated *input* and *output* parameters, analogous to procedures without bodies.

Messages are sent to tasks and received by them by *calling* and *accepting* entry points, respectively. A task may accept one of its own entry point through the `accept` statement, which is analogous to an on-site procedure declaration for the entry point; the programmer must specify an associated body which is passed the input and output parameters of the entry point, and is required to instantiate the output parameters before leaving the body. When a task `accept`s an entry, if another task isn't already suspended on *calling* that entry, then execution of the accepting task will be suspended that entry is called. Once a caller is established, the body of the `accept` statement is passed the input parameters and output parameters. Once the body is completed, the entry point call is completed, and the caller may resume.

Calling an entry point is done as though it were a regular procedure. If a task calls an entry point without the targeted task accepting that entry point, the caller's execution is suspended until the entry point becomes accepted and is handled.

Through *selective accept* statements, a task has the option to listen for multiple entries simultaneously, by providing an `accept` body for each one. The task resumes execution when the first of these entry points are called, at which point its corresponding body will be executed, and the *selective accept* statement completes. This means that once an accepted entry point inside a selective accept statement is called, the other `accept`s of the statement won't be (unless the statement is executed again).

Tasks may delay their execution through the `delay` statement – which delays task execution for a relative amount of time – or the `delay until` statement – which delays task execution until a specified point in time. Selective accept statements also supports using delays as selective accept candidates – allowing the task to time-out on waiting for messages and execute code to handle such time-outs.

3 The Basic Coroutine Design

The basic coroutine design was developed with three main goals in mind:

- To be as expressive as possible while still having the most common use cases be simple to express.
- Ensure that the design may be implemented such that all implicit state associated to elements of the design may be serialized with high *checkpoint compatibility*, as described in Section 2.2.2.
- Ensure that the total size of all allocations needed to preserve that implicit state while the simulator is outside of the modeled device – i.e.

not actively executing its behavioral code – has a statically determinable *finite upper bound*. In addition, the design should be tailored to ensure that – for any realistic usage pattern – the determinable upper bound of allocations needed should not dramatically exceed the amount typically used throughout the simulation.

In other words: the design must guarantee the existence of an implementation which may efficiently *statically allocate* all resources needed to preserve any coroutine-related state over the course of the simulation.

The design may still feature elements that would require unbounded dynamic allocation in order to be implemented; in this case, the design must ensure that any such resources *can be freed* by the time the device would return control to the simulation engine.

This goal is referred to as the *bounded state restriction*. It exists to simplify implementation of the basic design, and to ensure that the design may be implemented in such a way that resource leaks cannot occur over the course of a simulation.

This section describes each novel element introduced by the basic design; providing an overview of each feature, motivations for its inclusion and design choices made, brief specification, and possible implementation.

3.1 Coroutine Objects

3.1.1 Overview

Coroutine objects represent instances of launched coroutines. As a consequence, such instances are statically declared – new coroutines cannot be introduced during the course of a simulation.

A coroutine starts its execution at the beginning of the simulation, and its behavior is dictated through an associated ***`async`*** *method* (see Section 3.3).

Suspension and resumption of a coroutine are not done by referencing the coroutine object directly – instead, *channel objects* (see Section 3.2) are used as the underlying mediators between a coroutine and other components of the DML program.

3.1.2 Motivation

The choice to have coroutines be statically declared is in contrast to most general-purpose languages, where coroutines are typically created dynamically by means of a function which returns an object representing the coroutine.

This decision was made for two reasons:

(a) By having coroutines be statically declared, it's possible to implement them such that the required resources associated to them are also static.

This is needed to satisfy the bounded state restriction of the design, and dramatically simplifies serialization.

(b) DML is designed for modeling devices through declaring and referring to statically declared objects, which remain in scope for the entire lifespan of the simulation. Dynamically created coroutines are not only antithetical to this paradigm, but could also prove impractical, as it would make it difficult to refer to instances of dynamically created coroutines in static contexts (if not impossible).

3.1.3 Specification

DML is extended with a new compound object type, `coroutine`, which may appear as part of any object (including at top-level, as part of the device object).

Any declared coroutine object must be provided an `async` method of the following signature:

```
async method coroutine() -> ()
```

Coroutine objects have three main states: *unstarted/terminated*, *running*, and *suspended*. The execution of a coroutine is described by the `coroutine()` method of the object. This method is automatically called at the beginning of the simulation – as part of post-initialization – *starting* the coroutine – and when it terminates, the coroutine as a whole terminates.

A coroutine may become suspended as it executes an *await* expression (see Section 3.4) or a *race* or *concurrently* statement (see Section 3.5). Coroutines are resumed by being *sent* a message through a channel object that it is listening to, causing the coroutine to resume execution from its current suspension point. A resumed coroutine executes up until it next suspends itself, or terminates.

Coroutine objects offer built-in methods that allow a user to check which of the three main states a coroutine is currently in, as well as the `terminate()` and `restart()` methods that allow for the forced termination or restart, respectively, of a coroutine. `restart()` in particular is notable as it can be leveraged as part of the logic for handling a reset signal, or in order to restart a terminated coroutine. `restart()` terminates the coroutine if not already terminated, and then restarts execution from the beginning of the `coroutine()` method.

Coroutine objects offer built-in logic to automatically restart their execution when the parent device receives a reset signal. This logic can be overridden by instantiating the `sticky_coroutine` template or the `no_reset_coroutine` template – the former of which prevents the coroutine from being restarted upon soft resets, while the latter prevents the coroutine from being restarted upon any kind of reset.

The following is a simple example of a coroutine object, which makes use of channel objects and await expressions:

```
coroutine logger {
  channel uint64 log_chan;
  saved uint64 log_no = 0;

  async method coroutine() {
    while (true) {
      local uint64 to_log = await get_log_string;
      log info, 4: "logger: Log %lu: %lu", log_no, to_log;
      ++log_no;
    }
  }
}

method log_uint(uint64 to_log) {
  logger.log_chan.send(to_log);
}
```

3.1.4 Possible Implementation

Coroutine instances under the basic coroutine design may be compiled into *state machines*, and a coroutine object may represent all necessary resources and state associated to such an instance.

A coroutine object has two associated pieces of state which are serialized during a checkpoint: the *current suspension point of the coroutine* and *channel queue buffer nodes*, the latter of which are discussed in Section 3.2.4.

The current suspension point can be represented by a fixed-width integer – the specific width can be determined by the compiler, and can be as large as necessary in order to be able to represent each possible suspension point. This state is used by the associated `async` method of the coroutine in order to identify where execution should resume once the coroutine is resumed, and is modified to keep track of the new suspension point once the coroutine becomes suspended again. Suspension/resumption logic for `async` methods is discussed further in Section 3.3.3.

Although the suspension state as described in this section can be directly serialized, it is unsuitable for checkpointing – any change to an `async` method which would affect its suspension points would invalidate the number representing the current suspension point. In addition, the number itself isn't easy to interpret or modify in order to correct checkpointing issues that occur due to changes in the model. Instead of serializing the number directly, the current suspension point may be checkpointed by converting it to a representation of the callstack of `async` methods leading to the current suspension point. Deserialization thus functions by parsing this representation and producing the corresponding suspension point number. This representation

of the state is more tolerant to changes in the model by representing the suspension point relative to current callstack, and the representation allows it to be human-readable and modifiable. It is also possible to statically distinguish between kinds of suspension points – e.g. `race`, `concurrently`, or direct channel listens – allowing for improved readability and tolerance to changes in the model.

Simics serialization format bears similarity to JSON, in that serialized data may be represented through nested lists and dictionaries, together with values of primitive types such as strings, booleans, integers, and floating-point numbers [4]. Thus, one possible implementation for the approach above is to represent a suspension point through a three-element list:

$$[callstack, kind, index]$$

where

- *callstack* represents the callstack of `async` methods leading to the current suspension point, rooted at the `coroutine()` method. Each inner element of the callstack is itself a 2-element list representing an `async` method call:
  - The first element is a string uniquely naming the called method
  - The second element provides a 1-based index for which of the calls to the identified `async` method within the previous method on the callstack corresponds to the current suspension point.
- *kind* is a string representing the suspension point kind.
- *index* provides a 1-based index for which of the suspension points within the final method on the callstack of the identified suspension point kind is the current suspension point.

For example, consider the following code:

```
channel uint64 chan_A;
channel () chan_B;
channel () chan_C;

coroutine example_coroutine {
  async method coroutine() {
    while (true) {
      await foo();
    }
  }
}

async method foo() {
  local uint64 first = await chan_A;
```

```
  await bar(first);
  await chan_B;
  local uint64 second = await chan_A where (second > 10);
  await bar(second);
}

async method bar(uint64 i) {
  log info: "bar(%lu)", i;
  await chan_B;
  race {
    case (await chan_B);
    case (await chan_C);
  }
}
```

Starting from `coroutine()`, the suspension point corresponding to the `race` in the second call to `bar()` could be represented through:

```
[ [ ["foo", 1], ["bar", 2] ], "race", 1]
```

The suspension point corresponding to the second `await chan_A` in `foo` – the third direct channel listen within that method – could be represented through:

```
[ [ ["foo", 1] ], "listen", 3]
```

3.2 Channel Objects

3.2.1 Overview

Channel objects are the underlying mechanism through which coroutines are suspended and resumed. Any coroutine suspends itself by *listening* to a specified channel. At a later point, another part of the modeled device may *send* a set of values to that channel – called a *message* – which is then propagated to the coroutine, causing it to resume execution with access to the message sent.

Multiple coroutines may attempt to listen to a channel simultaneously, in which case any message sent to a channel will be propagated to each of those coroutines, which are then resumed one at a time; one coroutine receives the message and is resumed, and once it becomes suspended again the next coroutine receives the message and is resumed, etc. until all coroutines that listened to the channel have received the message and have become suspended again.

The basic design does not specify the order in which coroutines are resumed this way – however, the order must be deterministic; assuming the model remains unchanged, any message delivery following the resumption of a checkpoint must always exhibit the same behavior. A first-in, first-out (FIFO) order for message delivery is recommended: a message sent to a

channel will be delivered to listening coroutines in the order of least recently suspended. This would allow channels to act like a queue – indeed, this report uses the term *channel queue* to refer to the set of coroutines that are suspended by listening to a specific channel.

Channel objects must be declared by providing a tuple of type parameters to describe the type of messages sent through the channel. This tuple may be empty, in which case the channel is only used for suspension/resumption, and messages carry no information beyond the fact that they were sent.

The basic design specifies that channel objects are non-compound, but this is not by necessity; the basic design simply does not possess any elements would require channel objects to be compound, and non-compound objects allow for additional flexibility during implementation. Any extension to the basic design may choose to make channel objects compound if necessary.

3.2.2 Motivation

Channels are the result of attempting to design the underlying mechanism of suspension/resumption with two goals in mind:

(a) Be feature-rich enough to make their direct usage be suitable for most use cases. This is to address DML's relatively weak expressive power: if the basic suspension/resumption mechanism is too low-level, then it would become difficult to build abstractions on top of it within DML to represent even common use cases.

(b) Be powerful enough to allow their use as primitives in the implementation of other abstractions. In other words: the first goal must not make the mechanism too specialized to the point where arbitrary suspension/resumption of a coroutine would be impossible to implement.

To further elaborate on the first goal, channel objects were designed to make their direct usage suitable for:

- Modeling SystemC-like events, allowing coroutines to suspend their execution until a chosen set of events has occurred or until a specified timeout has occurred.
- Providing a means for coroutines to selectively receive and handle multiple different kinds of messages, similar to Ada selective accept statements.

A notable design choice of channel objects is that there is no native means for the *sender* to a channel to receive any output from the coroutines that the message was delivered to. This is in contrast to Ada, where task entries may have associated output parameters. Output parameters are fundamentally problematic within the basic design for multiple reasons:

- Unlike Ada entry calls, sends are *non-blocking*. This is by necessity – DML programs may contain logical components that both can't be arbitrarily suspended, and may need to communicate with coroutines. In particular, resumption of coroutines is expected to be mostly driven by interface methods implemented by the device, or by callbacks of triggered events.

  This makes it impossible to adapt the Ada design element that any entry call becomes suspended until the entry is fully handled – and thus, until all passed output parameters are guaranteed to be instantiated.

- Unlike entry points, channels may be shared between multiple coroutines; thus, each coroutine resumed by a send would instantiate the output parameters, resulting in multiple sets of output parameters. Resolving this issue is non-trivial, and would almost certainly complicate usage.

- Even if channels were restricted to only be used for one-to-one communication between coroutines, supporting an output value for a channel would severely complicate the coroutine design and its use. For instance, the design would need to ensure that any coroutine that receives a message through a channel must eventually provide all output parameters, such that the sender may resume. Ada serves as an existing reference as to how this could be done, but its approach can't be adapted directly – in particular, the execution of a coroutine can be canceled at any suspension point by the coroutine being externally terminated or restarted, thus making it impossible for a resumed coroutine to guarantee that it will be able to provide the output parameters.

Instead of providing built-in support for output parameters, users are expected to manually implement these ad hoc, thus moving the responsibility for safety and correctness to the user. For example, an output parameter that must be instantiated before the resumed coroutine suspends itself again can be implemented by passing a pointer as part of the message, with the convention that the resumed coroutine must write to the pointer before suspending itself.

3.2.3 Specification

DML is extended with a new non-compound object type, `channel`, which may appear as a member of any compound object. Channel objects can only be declared by also specifying a tuple of type parameters for the type of messages that may be sent through the channel. These type parameters will be referred to as `TPARAM_1, TPARAM_2, ...`. The following is an example of channel declaration, with the type of messages being the two-tuple `(uint8, bool)`:

```
channel (uint8, bool) example_chan;
```

A singleton tuple can be declared without parentheses:

```
channel uint8 example_chan;
```

And the absence of type parameters is declared through the empty tuple:

```
channel () example_chan;
```

Unlike other non-compound objects, channel objects have a number of associated built-in methods which may be called. These methods are specified below.

- `method send(TPARAM_1 msg_val_1, TPARAM_2 msg_val_2, ...) -> (uint64)`

  `send` traverses the set of coroutines suspended on the channel, and attempts to send the message `msg_val_1, msg_val_2, ...` to every coroutine in the set – resuming each coroutine that accepts the sent message, and removing them from the channel queue. The coroutines that reject the sent message remain in the queue. Once the final coroutine in the queue that accepted the sent message becomes suspended or terminates, `send` returns with the number of coroutines that accepted the sent message.

  The order in which suspended coroutines are delivered the message is not specified by the basic design, but is required to be deterministic.

  `send` will only attempt to deliver the message to the coroutines that were part of the channel queue at the beginning of the call to `send`. This means that if a coroutine resumed by `send` immediately suspends itself by listening to the same channel again, then the current execution of `send` *will not* send the message to that coroutine once more.

- `method suspended() -> (uint64)`
  `suspended()` returns the number of coroutines currently suspended by listening to the channel.

3.2.4 Possible Implementation

Channel queues may be implemented as linked lists, allowing delivery of messages to easily be implemented with FIFO semantics.

The necessary nodes for channel queues may be implemented as part of every coroutine object; every coroutine object contains a buffer of channel queue nodes, where each node may be part of a channel queue. Multiple nodes are needed per coroutine as a coroutine may listen to multiple channels simultaneously via `race` and `concurrently` statements. In order to statically allocate the buffer, static analysis is needed in order to determine the maximum number of channels a coroutine can simultaneously listen to – i.e.

the maximum number of `await` participants across all `race`/`concurrently` statements of the `async` methods in scope.

Every channel queue node in a coroutine contains the following information:

- A pointer to the parent channel corresponding to the channel queue. This may be `NULL` to indicate that the node isn't being used.
- A pointer to the next coroutine in the queue; `NULL` if the end of the queue has been reached.
- An index to identify which channel queue node of the next coroutine in the queue is used for the current channel queue.

Every channel object keeps track of the coroutines at the start and end of the channel queue, together with corresponding indices into the channel queue node buffers of those coroutines. The start pointer is needed as an entry point for queue traversals, and the end pointer can be used to efficiently insert newly suspended coroutines into the queue.

The responsibility of checkpointing channel queues falls *entirely* to the corresponding channel object; coroutine objects themselves *do not* serialize their channel queue node buffers. A channel object saves its state inside a checkpoint by traversing the channel queue and creating a corresponding list of the names of the coroutines suspended on the channel, in order. When restoring a checkpoint, the channel object reconstructs the queue by looking up the coroutines in the list by name, and modifying their channel queue node buffers. If the lookup of a coroutine fails, then that coroutine will not be included in the queue.

This serialization scheme has the benefit that despite the complicated run-time representation, the channel queue is exposed to the configuration environment in a centralized spot (an attribute of the channel object), which is easily accessible and human-readable – allowing for easy modification if necessary. In addition, the scheme is resilient to the addition and removal of coroutine objects, as well as the addition and renaming of channel objects.

The drawback with this scheme is that it is vulnerable to the renaming of a coroutine object, as that would cause the coroutine to lose its place in each channel queue it was in when restoring checkpoints – making it impossible to resume. This is detectable: an error can be logged following checkpoint restoration if the coroutine is suspended and its channel queue node buffer is empty.

This scheme is also vulnerable to the removal of channel objects; however, this issue is present in any serialization scheme, due to the fundamental issue that there is no satisfactory way to resolve a listen to a channel that doesn't exist anymore without manually changing the suspension state of the coroutine.

3.3 `async` Methods

3.3.1 Overview and Motivation

`async` methods are a new kind of methods that may only be called by a coroutine, and are able to perform operations related to that coroutine – most notably, suspending execution by listening to channels. `async` methods serve the same purposes as regular methods, but adapted for asynchronous code – they provide a means for code reuse, compartmentalization, and constructing interfaces.

`async` methods also have a number of restrictions in place to ensure that coroutine state may be serialized and represented statically – as is required by the restrictions placed upon the basic design. Notably, `async` methods are not allowed to be (mutually) recursive in order to be guarantee that the possible number of unique suspension points for a coroutine is finite, and `async` methods have unique scoping rules associated with `await` expressions to remove the need to store and serialize dynamically allocated resources.

3.3.2 Specification

`async` methods are declared exactly like regular methods, with the exception of an `async` prefix before the word `method`. `async` methods cannot be called using regular method call syntax, but must be called through an `await` expression. Unlike regular methods, `async` methods have the restriction that they may not be (mutually) recursive.

Unlike regular methods, the body of `async` methods are considered to be in *asynchronous scope*, meaning it is allowed to make use of

- `await` expressions;
- the `race` and `concurrently` statements.

When asynchronous scope does not apply to a block of statements – such as the body of regular methods – then that block is said to be in *synchronous scope*.

3.3.3 Possible Implementation

While coroutine objects store all necessary state corresponding to the state machine of a coroutine instance, `async` methods are used to describe the behavior of the coroutine at each state, as well as how the state machine should transition. `async` methods receive a parameter representing the suspension point they should *resume* from, and suspend themselves by returning the suspension point that they become suspended upon. Resumption of `async` methods is also accompanied with information about the delivery that caused the coroutine to become resumed. This information, together with the input

suspension point, can be studied by the generated function in order to jump to the correct point within the function using `goto`.

If an `async` method performs another `async` method call, then a range of suspension points is associated with that call, and resumption at a suspension point in that range will cause the generated function to `goto` to the `async` method call, and call the generated function of that `async` method with the corresponding suspension point. When that function returns, the parent `async` method will resume execution if the returned suspension point indicated that the called `async` method completed – otherwise, the calling `async` method will propagate the suspension to its own caller.

In order for this to be possible, it's necessary to be able to determine the number of suspension points within each `async` method called – including those from other `async` method calls these perform, in turn. This is done through static analysis – during compilation, a table may be built which associates every `async` method with the number of suspension points it contains. This analysis can also used to discover if there exists a mutually recursive chain of `async` methods, and the amount of storage needed to represent the highest number of suspension points among `async` methods.

3.4 `await` expressions

3.4.1 Overview and Motivation

`await` expressions are used to evaluate expressions that may potentially suspend the executing coroutine. There are two reasons why such expressions warrant the use of a new keyword:

- `await` expressions have unique properties, restrictions, and impact on surrounding code, and these aspects may be better communicated to the user by forcing them to preface the use of such expressions with `await`. Notably:
  - `await` expressions can only be used inside of `async` methods
  - Unlike any other kinds of expressions, `await` expressions may cause the simulation to progress between the beginning and end of the evaluation of such expressions.
  - `await` expressions cause local variables to be invalidated. (see Section 3.4.3)
  - `await` expressions can only be used in specific contexts; for example, they are not allowed as arguments to a method call.
- The `await` keyword makes it simple to define syntax for specialized variants of such expressions. An example of this is conditional channel listens (see Section 3.4.4).

3.4.2 Basic Specification

There are four kinds of await expressions:

- *`async` method call*: `await` *asyncmethodcall*
  where *asyncmethodcall* is an expression that is an `async` method call.

  This performs a call to the referenced `async` method with the provided arguments.

  `async` method calls are the only kind of await expressions that are not allowed to be used as triggers of participants inside of a `race` or `concurrently` statement.

- *Unconditional channel listen*: `await` *channelid*
  where *channelid* is an identifier for a `channel` object.

  This causes the executing coroutine to add itself to the channel queue of the referenced channel and suspend its execution until a message is sent to the coroutine via the channel. At that point, the sent message is *accepted*, causing the coroutine to be removed from the channel queue, and the await expression resolves and evaluates to that message.

- *Conditional channel listen*: Detailed in Section 3.4.4.

- *Delays*: Detailed in Section 3.4.5

`await` expressions may only be used inside of *asynchronous scopes* – i.e. within the body of `async` methods – and only in one of the following contexts:

- As a statement: `await signal_chan;`
- As the right-hand side of an assignment, including as an initializer for a variable declaration: `local uint8 packet = await get_packet();`
- As part of the trigger of a participant in a `race`/`concurrently` statement.
- As the expression component of a `return` statement:
  `return await get_packet();`

This restriction limits `await` expressions to contexts where their semantics can be unambiguously defined and easily understood, which also serves to simplify implementation. In particular:

- It ensures that coroutine suspension semantics is independent of evaluation order, removing the need to address the semantics of expressions such as `await get_number_A + await get_number_B`.
- It simplifies local variable invalidation (see Section 3.4.3) associated with the use of the `await` expression, removing the need to address how local variable invalidation applies to e.g. the following;

  ```
  local int foo = 4;
  local int bar = foo + await get_number();
  ```

3.4.3 Scoping Rules

The body of `async` methods are subject to a number of scoping rules, which are affected by the use of `await` expressions. These scoping rules limits the lifespans of any resources which are dynamically created during the execution of an `async` methods, such that any such resources become unavailable past any potential suspension point. Specifically, the scoping rules restrict the lifespans of local variables – i.e. variables declared `local` as well as method parameters. This ensures that any implementation of the coroutine design does not have to indefinitely store and serialize such variables – which would otherwise be necessary in order to preserve them between suspension points.

The scoping rules are as follows:

- Every local variable in scope *becomes invalidated* past any `await` expression, *unless* that identifier is a newly declared `local` variable which is initialized using the `await` expression.
- If the body of a loop statement – e.g. `while` or `for` statement – contains an `await` expression, then every local variable in scope *becomes invalidated* at the *beginning* of the loop statement.

  This rule is needed as local variables cannot be preserved past the first iteration of a loop containing suspension points.

Any attempt to use an invalid identifier past the point of invalidation *must* be rejected by the compiler. Beyond this, the design doesn't specify the exact definition of invalidation in this context; an abiding implementation may, for example, flag local variables as they become invalidated such that their further use is rejected, or have such identifiers exit scope entirely.

These scoping rules complicate usage of `await` expressions that evaluate to multiple output values – such as a listen to channels with multiple type parameters, or a call to an `async` method with multiple output parameters. The existing approach in DML to call methods with multiple output parameters is to declare the variables used to store output, and assign these to each output value of the called function:

```
method caller() {
  local uint8 first;
  local bool second;
  (first, second) = callee();
  ...
}
method callee() -> (uint8, bool) {
  ...
}
```

But this approach can't be used for `await` expressions, as these cause the declared variables to be invalidated. Two solutions have been identified:

- Relax the first scoping rule: a local variable becomes invalidated past any `await` expression, *unless* that `await` expression is the right-hand side of an assignment, and the local variable is a target of that assignment. This allows the conventional pattern described above to be used, at the cost of further complexity.
- Extend DML to permit the simultaneous declaration and initialization of multiple local variables, as follows:

  ```
  (local uint8 first, local bool second) = callee();
  ```

The basic design adopts the latter solution in order to avoid increasing the complexity of the scoping rules. The former solution is left as a possible extension.

3.4.4 Conditional Channel Listening

Conditional channel listening allow for coroutines to *selectively accept* a message sent to a channel, by specifying a condition that must be satisfied upon delivery. The syntax for conditional listens is as follows:

```
await channelid where (condition)
```

where *channelid* must be a identifier for a channel object, and *condition* must be a boolean expression. the semantics for a conditional listen is exactly that of an unconditional listen as detailed in Section 3.4.2 – except the message sent to channel may become *rejected*. Once a message is attempted to be delivered to the coroutine via the channel, the condition is evaluated, and if `true`, the delivery is considered accepted and the coroutine is removed from the channel queue and resumes execution as normal. Otherwise, the coroutine *rejects* the delivery, remains in the channel queue, and remains suspended at the same suspension point.

Local variable invalidation caused by the use of a conditional listen is already in effect for its conditional component; thus, the conditional component cannot reference local variables, with the exception of a newly declared local variable, if the conditional listen is used as the initializer for that variable.

If a conditional listen is used for an assignment, then any received message will be stored in the assign target *before* the condition is evaluated.

Conditional listens serve as built-in support for usage patterns where a coroutine may only resume execution following the reception of a message if a certain condition – possibly related to the received message – is met. For example, conditional listens allow the following:

```
local message_t message = await message_chan
                          where (message_ok(message));
```

which would otherwise need to be implemented through the following pattern:

```
session message_t message;
do {
  message = await message_chan;
} while (!message_ok(message));
```

which is both verbose and unintuitive due to the need for `session` variables as a means of circumventing local variable invalidation.

In addition, the inclusion of conditional listens permits additional support from other elements of the design:

- They may be used as triggers within `race`/`concurrently` statements.
- `send()` may exclude coroutines that reject the delivered message from the count of resumed coroutines that `send()` returns.

A motivating use case for conditional listens is to allow for coroutines to selectively accept messages sent through a channel used between multiple outstanding transactions. For example, let `message_t` be a type for a `struct` representing a message with a payload and an identifier for the specific transaction the message concerns, and `message_chan` be the channel for receiving such messages.

These could be defined as follows:

```
typedef struct {
  uint64 payload;
  uint64 target_tid;
} message_t

channel message_t message_chan;
```

Then a coroutine representing a specific transaction with transaction id `tid` could selectively wait for a message intended for it through the following:

```
local message_t message = await message_chan
                          where (message.target_tid == tid);
```

3.4.5 Delay

`await delay` expressions are a feature that closely resembles `after` statements, both in syntax and in purpose: `await delay` expressions are a convenience feature to allow for idiomatically suspending execution of a coroutine for a specified amount of time.

The syntax for `await delay` expressions is as follows:

```
await delay scalar unit
```

where *unit* must an identifier for a *unit of time*, and *scalar* must be a valid scalar for that unit.[5] For example, the following is a valid `await delay` expression:

```
await delay 1.2 s
```

The scalar does not have to be a constant expression, and, in fact, may reference `local` variables and method parameters; these are not invalidated until after the `await delay` expression.

`await delay` expression's most notable use case is as the trigger of a participant within a `race` statement, effectively allowing a coroutine to place timeouts on listens as needed.

`await delay` expressions may also be used without parameters, in which case these are called *immediate* ***`delay`*** *expressions*:

```
await delay
```

These are parallel to immediate `after` statements (see Section 3.6). An immediate `delay` suspends the executing coroutine and later resumes at the time the model would otherwise return control to the simulation engine – exactly as though the coroutine used immediate `after` to delay a send to a channel it then suspends itself upon. Just like immediate `after`, this can be used to control execution order: for example, a coroutine can use immediate `delay` following the reception of a message from a channel to ensure all other coroutines resumed by the message delivery act first. Immediate `delay` can also be used by a coroutine to suspend its execution until all pending immediate `after` statements have been resolved, which may be important for the coroutine's logic.

3.4.6 Possible Implementation

Outside of the triggers for participants in a `race`/`concurrently` statement, any channel listen generates a suspension point at that point in the `async` method. Listening to a channel follows the process described in Section 3.4.2, and if the coroutine becomes suspended to the channel, then the generated function of `async` method returns with the suspension point corresponding to the listen – as described in Section 3.1.4.

A conditional channel listen evaluates the condition component before proceeding past the suspension point. If rejected, then the coroutine remains in the queue, and suspends itself on the same suspension point again.

Each coroutine has an anonymous channel used for `await delay` expressions. Upon execution of an `await delay` expression, an anonymous event is posted with the specified scalar, and then the coroutine suspends itself

[5] Any adoption of the basic design must support the same units of time as for `after` statements. As mentioned in Section 2.3.7 this is limited to seconds only as of Simics 6.0.76.

upon the `await delay` channel. When the associated event is triggered, the callback sends a message to the channel, causing the coroutine to resume execution. In the case of immediate `delays`, the message is delayed via the immediate `after` queue instead (see Section 3.6.3).

Multiple `await delay` expressions as triggers within a `race`/`concurrently` can be supported by having each message sent to the anonymous `await delay` channel identify which of the `await delay` triggers the message corresponds to.

In order to correctly handle termination of the executing coroutine, or cancellation if used as the trigger of a participant inside of a `race`/`concurrently` statement, the cancellation of any `await delay` expression will cause the posted event or entry in the immediate `after` queue associated with the `await delay` to be removed.

3.5 The `race` and `concurrently` Statements

3.5.1 Overview and Motivation

With only simple `await` expressions, coroutines would be unable to react to multiple different kinds of events at a suspension point, as they would be restricted to listening to one explicit channel, indefinitely, unable receive messages from any other channel. This would be an unacceptable shortcoming – coroutines are intended to replace traditional state machines for the purposes of developing asynchronous logic, and thus it is important to be able to react to multiple different kinds of messages, as well as be able to establish timeouts for any listen. A feature mirroring that of Ada's *selective accept* statement is needed.

Another desirable feature would be the ability to wait on multiple channels simultaneously and resume execution only once a message has been received for each channel – as is possible with SystemC events.

The basic coroutine design addresses these needs through the `race` and `concurrently` statements, which allow the user to specify a number of `await` expressions to be executed simultaneously, and for each such expression, an associated (compound) statement to be executed once that await expression terminates. Each `await` expression that are simultaneously executed are called *triggers* of the `race`/`concurrently`, and the trigger together with its associated body is called a *participant*.

`race` and `concurrently` statements differ in when they complete execution. A `race` statement completes when *any* participant is triggered, while `concurrently` statements complete when *every* participant has been triggered. `concurrently` is intended to be used to represent events which all need to be triggered in order for the coroutine to resume execution, while `race` is intended to be used for handling different kinds of messages, or to establish timeouts or handle abnormal events.

`race` and `concurrently` can also be used to declare groups of participants inside of `race` and `concurrently` statements, and their roles reflect the statement of the same name; a `race` participant completes once any of its immediate participants complete, and a `concurrently` participant completes once all of its immediate participants complete. Such groups can be arbitrarily nested.

3.5.2 Specification

Two new statements, `race { ... }` and `concurrently { ... }` are introduced, and may only be used within asynchronous scopes. The body of a `race` or `concurrently` statement must contain one or more participant declarations. There are three kinds of participants: `await`s, `race`, or `concurrently`. When declaring a participant, it may be optionally prefaced with a *guard*, through `if (`*guard*`)`, where *guard* must be a (non-`await`) boolean expression.

An `await` participant is declared through the following:

`case (`*trigger*`)` *body*

where

- *body* is a statement (typically a compound statement or `;`)
- *trigger* must either be an *expression*, *assignment*, or *local variable declaration*, and must contain exactly one `await` expression.

The trigger of an `await` participants are subject to the following restrictions:

- The `await` expression must not reference a channel used inside of the trigger of another `await` participant within the same `race`/`concurrently` statement.

  This is to avoid the question of what the semantics should be when multiple participants wait on the same channel – an issue called *duplicate listening*, discussed in Section 3.8.2.

- The `await` expression of the trigger *must not* be an `async` method call.

  Issues related to such triggers are discussed in Section 3.8.3.

`race` and `concurrently` participants are declared exactly like `race` and `concurrently` statements, respectively (with the exception that as participants, they may be prefaced with a guard).

The following is an example of a valid `race` statement, assuming all relevant identifiers are in scope.

```
race {
  concurrently {
    case (local message_t message = await get_message
```

```
          where (message.target_tid == active_tid)) {
      received_payload = message.payload;
    }

    if (!progress_permitted) case (await progress_permission);
  }

  if (have_timeout) case (await delay transaction_timeout s) {
    got_timeout = true;
  }
}
```

The detailed semantics of the execution of a `race`/`concurrently` statement is covered in Appendix B. In summary:

- The guards of every participant are only evaluated once, at the beginning of the `race`/`concurrently` statement. These are used to determine the initial participants of the `race`/`concurrently` statement. The coroutine then performs all setup necessary such that it becomes notified once any trigger is completed.
- Once notified of a trigger being resolved successfully, any other remaining participants which are invalidated as a result are *canceled* and removed from the `race`/`concurrently`. Any resources associated to the triggers of the canceled participants are cleaned up; for example, the coroutine removes itself from the queue of each channel referenced in the invalidated triggers.

  Following cancellation, the body of the triggered participant is executed.
- If no valid participants remain, the `race`/`concurrently` statement is completed, and execution resumes past it. Otherwise, the coroutine suspends itself again.

Like `await` expressions, `race`/`concurrently` statements have scoping rules associated with them in order to limit the lifespans of local variables.

(a) Every local variable in scope *becomes invalidated* past any `race`/`concurrently` statement.

(b) If the body of a loop statement – e.g. `while`, `for` or `foreach` statement – contains a `race` or `concurrently` statement, then every local variable in scope *becomes invalidated* at the *beginning* of the loop statement.

(c) For every `await` participant (either immediate participants of the top-level statement or within nested `race`/`concurrently` participants), the scope in place for the associated *guard* (if present), *trigger* and *body*

components behaves exactly as though the entire `race`/`concurrently` statement was replaced with the following:[6]

```
if (guard) {
  trigger;
  body
}
```

This means:

- Any valid local variables at the beginning of the `race`/`concurrently` statements are still considered valid for the purposes of evaluating the *guard* and *trigger* components.
- If *trigger* is a variable declaration, then that variable is available in *body*, but any other `local` variables or method parameters are invalidated.
- Any variables declared in *trigger* or *body* will not be in scope outside of the `await` participant.

In addition to the above, `race`/`concurrently` statements have a scoping rule which restricts the use of `await` expressions within them:

For every `await` participant, if that participant is *not* an immediate or indirect descendant of a `concurrently` statement or participant, then the body of the `await` participant is in asynchronous scope. Otherwise, the body is in synchronous scope.

For example:

```
race {
  race {
    case (await chan_A) {
      ... // In asynchronous scope
    }
  }

  concurrently {
    race {
      case (await chan_B) {
        ... // In synchronous scope
      }
    }
  }
}
```

This rule makes a compromise between the heavily desirable property for bodies of `await` participants to be asynchronously scoped, and the untenable semantic difficulties that would present with `concurrently` statements.

[6] If the *guard* component is omitted, then the same applies except *guard* is replaced with `true`.

One of the major use cases for `race` statements is to receive and handle multiple different messages – such uses will take the form of `race` statements with only top-level `await` participants. In these cases, the desired behaviour is that the race ends immediately once a top-level `await` participant of a race is triggered – that way, execution can resume inside the body of that participant, and may use `await` expressions without fear of being interrupted by a *different* participant in the race being triggered. The semantics for this is easy to define and implement, and thus, such participants should be considered asynchronously scoped.

However, for any other kind of `await` participant, being triggered doesn't necessarily mean that the `race`/`concurrently` statement is completed – there may be other participants that need to be triggered in order for the statement to complete. This makes it problematic to allow the body of such participants to be asynchronously scoped – if the body executes an `await` statement and suspends itself, should the participant still be considered part of the race, even though its trigger has completed? Not only would that contradict the desirable behavior for immediate `await` participants of `race` statements as detailed above, but it would also severely complicate both specification and implementation of the design. Thus, such participants should only be considered synchronously scoped.

The above scoping rule identifies all participants that are *guaranteed* to result in the race being completed once triggered – and thus, whose bodies can safely be in asynchronous scope. This resolves the tension between `race` and `concurrently` statements, at the cost of increased complexity.

3.5.3 Possible Implementation

Any `race`/`concurrently` statement generates only one suspension point, no matter how many participants are within it.

When the coroutine is resumed from that resumption point, the channel which caused resumption is identified, and the code tries to resolve the trigger that's waiting on it. If the sent message is accepted, the channel is removed from the coroutine's channel queue nodes, and the check is performed for what participants should be considered completed. This may lead to cancellations of other participants.

Following that check, the body of the triggered participant is executed. Once that body completes, if the check determined that the `race`/`concurrently` statement as a whole is completed, then execution moves to past the `race`/`concurrently` statement. If not, then the `async` method suspends itself with the suspension point of the `race`/`concurrently` statement.

The check for what participants should be considered completed can be done by traversing the channel queue nodes of the executing coroutine to determine what channels the coroutine is still suspended upon – from that, it's possible to deduce which `await` participants still remain in the race,

which can be used to determine the status of all other participants, as well as the `race`/`concurrently` statement itself.

3.6 The Immediate `after` Statement

3.6.1 Overview and Motivation

The immediate `after` statement is an extension to the existing `after` statement, and allows for specifying a method call to be executed once control would next be returned to the simulation engine.

This is useful to avoid issues that stem from asynchronous code causing a message to be sent to a channel *synchronously* before the asynchronous code has had the opportunity to listen to that channel. Consider the following example device, which both `connect`s to and `implement`s a simple `ping` interface in order to communicate with another device:

```
connect ponger {
  param configuration = "required";
  interface ping;
}

implement ping {
  method ping() {
    ping_channel.send();
  }
}

channel () ping_channel;
saved bool gotten_pong = false;

coroutine pinger {
  channel () should_send_ping;

  async method coroutine() {
    await should_send_ping;
    ponger.ping.ping();
    await ping_channel;
    gotten_pong = true;
  }
}

method send_ping() {
  pinger.should_send_ping.send();
}

method gotten_pong() -> (bool) {
  return gotten_pong;
}
```

The intention is that once `send_ping()` is called, the modeled device – the *pinger* – sends a ping to the connected device – the *ponger* – and waits for a response – and once the response is received, the coroutine ensures `gotten_pong()` returns `true`.

The code may seem simple, but it contains a bug: the *ponger* may send its response *synchronously*, as part of the `ponger.ping.ping()` call. This will cause the `implement` object to send a message to the `ping_channel` before the coroutine has had the opportunity to receive it via the `await ping_channel;` statement – breaking the logic of the code.

The solution is to ensure that the coroutine is ready to receive the message in time, by delaying sending the message until the current line of execution is resolved, and control would be given back to the simulation engine. This is exactly what the immediate `after` statement allows.

The bug can be addressed by modifying the `implement` object as follows:

```
implement ping {
  method ping() {
    after: ping_channel.send();
  }
}
```

3.6.2 Specification

The immediate `after` statement is invoked as follows:

```
after: callback;
```

where *callback* is any valid (non-`async`) method call. This in contrast to regular `after` statements, which only support method calls without parameters.

The immediate `after` statement causes the callback to be executed once control would otherwise return to the simulation engine. If multiple immediate `after` statements have been invoked at that point, then each corresponding callback is executed in order. Any such callback may execute further immediate `after` statements, which will lead to the callbacks of these to be invoked the current set of immediate `after` statements have finished resolving. This process continues until all pending callbacks of immediate `after` statements have been resolved, at which point control returns to the simulation engine.

The parameters to the method call are evaluated at the point the `after` statement is invoked. This makes it necessary to store the evaluated parameters until the callback is executed. This does *not* contradict the second or third goals of the basic design – control is never returned to the simulation engine between the invocation of an immediate `after` statement and the resolution of its callback, meaning any dynamically allocated resources needed to execute immediate `after` statements can be freed before execution leaves the device and the simulation resumes. Consequently, these resources do not

need to be serialized either, as checkpointing can only occur once control has been given to the simulation engine.

3.6.3 Possible Implementation

Unlike other elements of the basic design, the immediate `after` statement requires extending the Simics simulator itself, and not just the DML compiler. Simics offers a number of hooks to particular events concerning the execution of the simulator itself to which callbacks may be attached. In order to perfectly satisfy the specification for immediate `after` statements, a hook is required for when control is returned to the simulation engine from any device model. If provided, immediate `after` statements can be implemented through the following:

- There exists a simulation-wide immediate `after` queue.
- Any immediate `after` statement causes a closure for the specified callback and provided parameters to be added to the immediate `after` queue.
- A global callback is established such that once control would be returned to the simulation engine, the closures of the immediate `after` queue are repeatedly dequeued and executed until the queue is empty, at which point control returns to the simulation engine.[7]

A simulation-wide queue, and a global hook for control being returned to the simulation engine from *any* device could pose significant implementation issues. An alternate approach which slightly violates the specified semantics of immediate `after` statements would be to have device-local immediate `after` queues – which are easily implemented as part of each generated device model – and to rely on hooks for control entering or exiting specific devices. If this was available, then the following approach could be used to implement immediate `after` statements:

- Every device has an associated immediate `after` queue, and a *depth* counter.
- Any immediate `after` statement causes the specified callback to be added to the device's immediate `after` queue.
- A callback is attached to the device entry hook, which increments the device's depth counter.
- A callback is attached to the device exit hook, which decrements the device's depth counter. If the depth counter reaches 0, then the

[7] If any immediate statements are executed as a result, the callbacks will be enqueued and subsequently dequeued before control returns to the simulation engine.

callbacks of the immediate `after` queue are continuously dequeued and executed until the queue is empty.

This causes callbacks queued with immediate `delays` to be delayed only until all pending entries to the parent device have completed, which does not always correspond to execution being returned to the simulation engine. This difference is insignificant for the intended use case of immediate `after` statements; the ability to control execution order of statements *within the scope of a specific device model*.

To demonstrate why the depth counter is necessary, consider the following scenario:

- The simulation engine enters device A. The device A entry hook is triggered.
- Device B is called from device A via a connection.
- Device A is called from device B via a connection. The device A entry hook is triggered.
- Device A executed an immediate `after` statement and returns control to device B. The device A exit hook is triggered.
- Device B returns control to device A.
- Device A returns control to the simulation engine. The device A exit hook is triggered.

Without the check for device entry depth, the immediate `after` queued by device A would be immediately executed once control is returned to device B, despite the fact that there is a device entry to A still incomplete.

3.7 Problematic Interactions

The basic coroutine design admits several interactions that have been identified as problematic in ways that are particularly difficult to resolve. These interactions are therefore considered *forbidden*, and may cause an error to be logged, or may even be considered undefined behavior. Forbidden interactions are as follows:

(a) The `terminate()` and `restart()` methods of a coroutine object may not be called while the associated coroutine is running – it must be suspended or terminated. In other words, a coroutine may not call `terminate()` or `restart()` upon itself as part of its asynchronous logic.

This interaction is problematic as it raises the question of how the execution of the coroutine should proceed following such a call. The most intuitive semantics would be for the coroutine to immediately

abort the current line of execution and either terminate or move execution to the start of its logic (`terminate()` and `restart()`, respectively). However, these semantics are considered unsatisfactory, as they would give rise to subtle inconsistencies which would present unacceptable bug sources. To elaborate, consider the following method:

```
method restart_all_coroutines() {
  coroutineA.restart();
  coroutineB.restart();
}
```

With the semantics above, if `restart_all_coroutines()` were to be called inside of `coroutineA`, only that coroutine would be restarted; the second statement is never executed as `coroutineA` immediately aborts the line of execution when restarted. In contrast, `restart_all_coroutines()` would restart both coroutines if called inside `coroutineB` or outside either coroutine.

(b) While a coroutine is evaluating the trigger of or executing the body of a participant within a `race`/`concurrently` statement, it is forbidden to send a message to a channel which is referenced within the trigger of a remaining participant of the `race`/`concurrently` statement.

This is problematic as although the coroutine is present within the referenced channel queue, it is not in a state in which it is capable of handling the delivered message.

Note: This only applies if the coroutine is actively running by the time the message is sent. A simple way to guarantee that it is not is by delaying the send using the immediate `after` statement.

In addition, remaining participants are determined when a trigger is resolved, and a trigger being resolved may cause multiple participants to exit a `race`/`concurrently` statement. For example, the body of an `await` participant directly in the body of a `race` statement is not affected by this rule, as its trigger being resolved will cause all other participants to exit the race.

(c) While the conditional component of a conditional listen is being evaluated, it is forbidden to send to the channel that is being conditionally listened to – i.e. evaluating the expression for the conditional component must not cause a send to that channel.

This is problematic as it results in a circular dependency which is impossible to resolve: the send already in progress can't be completed before executing the new send – whose semantics would rely on if the coroutine accepted the send already in progress.

An additional interaction has been identified as problematic, but is possible to resolve in a partially satisfactory manner: the action of calling `send()` of a channel while an existing `send()` of the same channel is already being executed. This is problematic as it would result in multiple sends to the same channel to be executed concurrently.

A consequence of forbidding this interaction would be that any coroutine that is resumed by the result of a sent message *must not* synchronously send another message to the same channel before the coroutine's next suspension point. This can be addressed by delaying sending the follow-up message through the immediate `after` statement.

One approach to support this interaction would be to have calls to `send()` *resume and complete* any traversal of the channel queue already in progress from an existing call to `send()`, and *then* perform the process of sending the new message. A consequence of this approach is that if yet another `send()` of the same channel is performed as part of completing the existing traversal, then that `send()` must be completed before the existing traversal can be considered completed. This means executions of `send()` calls chained this way are completed in a *Last-In First-Out* order.

3.8 Notable Avenues for Improvement

3.8.1 Dynamically Created Coroutines

Section 3.1.2 details two motivations for the decision to make coroutine instances statically declared:

(a) In order to offer the opportunity to statically allocate resources for coroutine instances, as is required by the bounded state restriction.

(b) In order to follow the general DML paradigm of statically declared objects. Dynamically created coroutines would lead to difficulties as they can't be statically referenced.

However, each of these motivations may reasonably be questioned:

(a) does not apply if the unbounded state restriction were to be lifted. Although serialization would become more difficult – as instances of launched coroutines would no longer be statically known – it would not be impossible. Device attributes must be static, but the serialized representation can be dynamic in size – thus, a single attribute can be responsible for serializing a pool of dynamically created coroutines.

(b) Almost no elements of the design require explicitly referencing coroutine objects – most communication is done indirectly through channel objects. The most notable exception is for the methods that coroutine objects provide, and the use cases for these could possibly be replaced

with other features. Therefore, it's plausible that dynamically created coroutines could be supported with minimal tension, unless further extensions are made which rely on explicitly referencing coroutine instances.

Because of these reasons, a potential avenue for improved flexibility in the design is to develop support for dynamically created coroutines to some extent.

3.8.2 Duplicate Listening

The design restricts `race/concurrently` statements such that a coroutine may not perform multiple simultaneous listens of the same channel – called *duplicate listening* for short. A possible extension to the design would be to allow duplicate listening – allowing multiple triggers to reference the same channel – and define the semantics for it.

Support for duplicate listening is a prerequisite for allowing `async` method calls as triggers, as such `async` methods may listen to the same channels that the coroutine is already waiting upon in the parent `race/concurrently` statement.

Suggested semantics for duplicate listening will not be presented in this report, due to several difficult unresolved difficulties. In particular, when multiple participants of a `race/concurrently` statement are triggered simultaneously, which of the participants' bodies should be executed; and if multiple bodies are to be executed, how is this done?

3.8.3 `async` Method Calls as Triggers

`async` method calls are not permitted as triggers within `race/concurrently` statements. Such triggers would be problematic for three reasons:

- Multiple `async` method call triggers would mean that the coroutine needs to maintain multiple concurrent lines of execution – each with separate suspension states. The semantics of this are not difficult to resolve, but implementing it under the bounded state restriction would be.
- Such method calls may listen to channels already referenced in the parent `race/concurrently` – leading to duplicate listening. The semantics of duplicate listening are difficult to resolve, and are further complicated by the fact that they can occur due to two separate points of execution simultaneously maintained by the same coroutine.
- `race/concurrently` statements require some associated state. By ensuring a coroutine may only perform a single such statement at a time,

the state necessary for such statements can be efficiently statically allocated through the means described in Section 3.5.3. However, `async` method calls as triggers would allow a coroutine to execute multiple `race`/`concurrently` statements simultaneously – making implementation under the bounded state restriction problematic.

Any variant of the design which lifts the bounded state restriction while also supporting duplicate listening would, in theory, be able to support `async` method calls as triggers.

4 Iteration

The basic design is used as the basis for the development of two forks of the design, called the *bounded design* and *unbounded design*.

- The bounded design continues the development of the basic design, and has all the same goals – including the enforcement of the *bounded state restriction*. This design will – if necessary – extend upon the basic design with ad hoc solutions that would prove redundant in the unbounded design.
- The unbounded design has all the same goals of the basic design, with the exception of the *bounded state restriction*, allowing for the inclusion of design elements that would be forbidden otherwise. This design prioritizes usability before ease of implementation and safety; in particular, it does not guarantee the absence of resource leaks.

These design are initially identical to the basic design, but are separately *iterated upon* by identifying issues of the design – usability flaws, semantic issues, and feature gaps – studying possible solutions, and implementing these into the designs. This section details the changes to each design that were made throughout the iteration process by discussing each issue identified and how it was addressed.

Several identified issues of the design were found to have related solutions relying on a particular usage pattern called *subcoroutines*. Due to the evident power of this usage pattern, it became a major influence throughout the iteration phase – in particular, usability issues of subcoroutines became an issue that needed to be addressed.

Section 4.1 covers the core usage pattern of subcoroutines and usability issues of this pattern. All other subsections detail a particular issue and discusses how it may be addressed in each design: Section 4.2 covers boilerplate issues of standard coroutines and how these may be addressed, which influences the choices made in Section 4.3 to address the usability issues of subcoroutines; Sections 4.4 and 4.5 detail issues either resolved by or heavily related to the use of subcoroutines; and Sections 4.6 through 4.8 cover issues largely unrelated to subcoroutines.

4.1 Subcoroutines

As the iteration phase progressed, many identified issues of the design were found to have related solutions which relied on coroutines leveraging other coroutines created specifically for the purposes of resolving the issue. Such coroutines are called subcoroutines, identified by the fact that their execution – their start and potential cancellation – is controlled by another coroutine, known as the subcoroutine's *parent*. Subcoroutines typically correspond to an asynchronous subroutine call, with associated input and output. Input parameters are provided when the subcoroutine is started, and output parameters are delivered to the parent upon completion.

Subcoroutines can be implemented in the basic design, through the following pattern:

- The subcoroutine instance is represented by a coroutine object, possibly a subobject of its parent coroutine.
- The subcoroutine features a `start` channel parameterized with the input parameters, and potentially a `done` channel parameterized with the output parameters. The `start` channel is used to control the start of the subcoroutine's execution, and the `done` channel is used to signal completion and provide output parameters to the parent.
- If necessary, the subcoroutine may declare mutable variables in order to store output parameters and signal completion, in case the parent coroutine is not prepared to handle completion of the subcoroutine at the time the subcoroutine becomes completed.
- If necessary, the subcoroutine is configured such that their execution is not restarted upon reset signals (as the subcoroutine should only be canceled by its parent).
- If on-site cancellation handling is necessary (see Section 4.5), then it is necessary to declare a `canceled` channel used by the subcoroutine to handle cancellation, and define a `cancel()` method for use by the parent coroutine.

However, this pattern has two notable issues:

- The declaration and use of subcoroutines involve significant amounts of boilerplate.
- A subcoroutine declaration only represents *one* instance; any declared subcoroutine can't safely be used by multiple coroutines simultaneously. This makes it necessary to duplicate any subcoroutine declaration for each coroutine that needs to make use of it.

DML's metaprogramming features can be leveraged in order to easily duplicate any declarations in a compact manner; the primary issue is thus the basic boilerplate needed to declare and use a subcoroutine. Due to the evident power of subcoroutines, resolving these issues became a major goal as the iteration phase progressed.

4.2 Concise Coroutine Declarations

The boilerplate incurred from declaring coroutines can become relatively heavyweight when the behavioral logic of the coroutine itself is simple. In particular:

(a) For all use cases of coroutines identified, it is desirable to have their logic *loop forever*. This must be done explicitly through a `while (true)` statement, incurring additional boilerplate.

(b) The behavioral logic of a coroutine cannot be specified directly with the declaration of the coroutine object, but must be specified by defining the `coroutine() async` method within the object's body.

(c) Due to the above, the behavioral code for the coroutine instance is typically placed at three levels of indentation.

(d) Although channels may be shared between coroutines, it is predicted that simple coroutines only make use of channels exclusive to it. In particular, it is predicted that the start of a simple coroutine would almost always be controlled through a single channel declared specifically for that purpose. The declaration and use of that channel incurs additional boilerplate.

Most coroutine declarations can be expected to have boilerplate incurred from the above, which thus presents a usability issue.

4.2.1 Implicit Looping of `coroutine()` Method

A solution to Issue a is to have the `coroutine()` method of a coroutine object implicitly *loop forever*; once terminated, it is immediately called again. This makes it unnecessary for a user to manually implement such looping, removing that source of boilerplate, and reduces the expected levels of indentation for specifying behavioral logic for a coroutine instance to two levels.

This change would have a number of consequences and issues:

- The design no longer provides any intrinsic means for executing coroutines to place themselves in a terminated state.

- Due to the above, the *terminated* coroutine state becomes significantly less important – outside of external termination via `terminate()` calls, it'd be impossible for a coroutine to be placed in the terminated state once it has been started during post-initialization. Indeed, it would be more appropriate to call the state *unstarted* with this change. As a coroutine cannot naturally terminate, the native support for external termination via `terminate()` now presents an inconsistency, and `terminated()` becomes almost meaningless.
- The fact that `coroutine()` implicitly loops forever is unintuitive, inconsistent with the behavior of other `async` methods, and may present a source of bugs. In particular, if the `coroutine()` method of a coroutine is defined with an empty body, then post-initialization will never complete as the coroutine will never become suspended.

These may be addressed as follows:

- A terminated state can manually be implemented by having a coroutine listen to a channel exclusive to it which never receives a message – called a *termination channel*.
- `terminate()` and `terminated()` are no longer valuable if a coroutine can no longer become naturally terminated, and should thus be removed.

  If external termination of a particular coroutine is desirable, it can be implemented through the following scheme:

  - Have a `session` boolean variable be part of the coroutine object. This is used as a flag, studied by at the start of `coroutine()` in order for the coroutine to determine if it should place itself in a terminated state by listening to a termination channel following a `restart()`.
  - `terminate()` can now be implemented by setting the flag, calling `restart()`, and then clearing the flag.

  `terminated()` can be implemented by studying the number of coroutines suspended on the termination channel.
- Although the fact that `coroutine()` implicitly loops forever is unintuitive, it can easily be learned and understood as long as it is properly documented.

  The ease of which a `coroutine()` declaration can result in an infinite unyielding loop can be ameliorated by forbidding definitions of `coroutine()` that can be statically determined to never suspend. A simple means for doing so is for the compiler to reject any definition of `coroutine()` with a number of associated suspension points equal to

zero. In particular, this prevents an empty body from being a valid definition of `coroutine()`.

As all issues with the change can be satisfactorily resolved, and the change significantly reduces overhead for simple uses of coroutines, it is incorporated into both the bounded and unbounded designs.

4.2.2 Unifying Coroutine Objects and `coroutine()`

One approach to resolving Issue b is to unify the declaration of a `coroutine` object and its associated `coroutine()` method. That is, instead of the following:

```
coroutine coroutinename {
  async method coroutine() {
    body
  }
}
```

Coroutines would be declared as follows:

```
coroutine coroutinename {
  body
}
```

This means coroutine objects would no longer be compound. The most significant loss this would present would be *lack of configurability*; any chosen behavior upon the various forms of built-in resets must be built into the coroutine type itself, and can't be overridden.

If the default behavior that coroutines restart execution upon resets is kept, this change would make the design *strictly less general*, as then that behavior can't be disabled. Thus, in order to preserve generality, coroutines under this design must *not* restart execution upon resets by default – instead, that behavior must be specified by the user, e.g as follows:

```
coroutine coroutinename {
  body
}

method hard_reset() {
  coroutinename.restart();
}
```

As this behavior is expected to be the most commonly desirable, requiring the user to specify it could result in the change *increasing* boilerplate, rather than reducing it. However, this is only relevant if the parent device makes use of a built-in reset signal.

Because of this, a conclusive answer has not been reached as to what design is more desirable. This will be determined during the evaluation phase

– in particular, it is necessary to study how large proportion of applicable use cases for coroutines also make use of built-in DML reset signals.

4.2.3 Specialized Syntax for Starting Coroutines

Issue d can be addressed by providing a built-in means for controlling the execution of the main logic of a coroutine, rather than relying on an infinite loop where each iteration is separated through a listen to a channel.

Using the design presented in Section 4.2.2 as a basis, one possible method to support this would be to allow a coroutine to specify input parameters, and allow the DML program to start execution of the coroutine by providing those parameters through a built-in `start` method. This design increases the significance of the *terminated* coroutine state, thus justifying the reintroduction of the `terminate()` method, as well as `terminated()` or an inverse thereof – `started()`. The latter is considered to be intuitive as coroutines are no longer started at the beginning of the simulation, making the *unstarted/terminated* state any coroutine's default state – being idle.

In addition, `restart()` must be modified to require the input parameters of the coroutine – as these must provided at the start of the coroutine logic.

To exemplify this design, the following declaration:

```
coroutine sample_coroutine(uint8 input_1, bool input_2) {
  body // input_1 and input_2 are in scope as local variables
}
```

would be essentially equivalent to the following under the current bounded design:

```
coroutine sample_coroutine {
  // Template to prevent coroutine from being restarted upon any reset.
  is no_reset_coroutine;

  channel (uint8, bool) start_chan;

  async method coroutine() {
    (local uint8 input_1, local bool input_2) = await start_chan;
    body
  }

  method start(uint8 input_1, bool input_2) {
    start_chan.send(input_1, input_2);
  }
}
```

The drawback with this approach is that by elevating one particular form of control over the start of a coroutine, all other forms become significantly more difficult to implement. For example, it may be desirable to have the start of a coroutine rely on a channel shared between multiple coroutines; or a

conditional listen; or a `race`/`concurrently` statement. In order to implement such a coroutine, it is necessary to manually start the relevant coroutine at post-initialization, such that regular suspensions may be used to control the effective start of the coroutine logic instead of the built-in mechanism provided by the design. This leads to significant amounts of non-intuitive boilerplate.

For example, the following code under the current bounded design, keeping coroutine objects compound:

```
channel uint8 chan_A;
channel bool chan_B;

coroutine sample_coroutine {
  async method coroutine() {
    race {
      case (local uint8 msg_A = await chan_A) {
          bodyA
      }

      case (local uint8 msg_B = await chan_B) {
          bodyB
      }
    }
    bodyC
  }
}
```

must be implemented as follows in the proposed design, *not including* any boilerplate necessary to handle resets:

```
channel uint8 chan_A;
channel bool chan_B;

// Pseudobank; does not contain any registers
bank sample_coroutine is (post_init) {
  method post_init() {
    // Post-initialization occurs both at the start of a simulation
    // and as part of the resumption of any checkpoint.
    // If the latter, then the coroutine should not be attempted to be
    // started. This can be checked via a Simics API call.
    if (!SIM_is_restoring_state(dev.obj)) {
      inner.start();
    }
  }


  coroutine inner() {
    while (true) {
      race {
```

```
        case (local uint8 msg_A = await chan_A) {
            bodyA
        }

        case (local uint8 msg_B = await chan_B) {
            bodyB
        }
      }
      bodyC
    }
  }
}
```

The plausible need to circumvent what is otherwise intended to be a feature – the explicit, parameterized start of coroutines – demonstrates that the presented design is not a viable alternative to the existing design.

4.3 Support for Idiomatic Subcoroutines

As previously noted, subcoroutines are a powerful, but boilerplate intensive pattern. Because of this, it is heavily desirable to extend upon the bounded and unbounded designs with native support for subcoroutines.

4.3.1 Bounded Design

Section 4.2.3 showcases a design for coroutines with built-in support for input parameters, which was rejected as a replacement for the existing design due to its lack of generality. However, this design is well suited for adaptation into an extension of the bounded design in order to idiomatically declare subcoroutines through *subcoroutine objects*. The execution of a subcoroutine is entirely controlled by its parent; making their desirable execution behavior identical throughout all uses. Thus, the lack of configurability does not pose an issue:

- The start of a subcoroutines is always explicitly initiated by its parent coroutine – as such, they are naturally restricted such that there is only a single point of entry, which can't be rejected. All other forms of starting a coroutine – shared channels, `race`/`concurrently` statements, conditional listens – are not relevant for subcoroutines, and do not need to be represented.

- The execution of a subcoroutine should only be canceled by its parent; in particular, subcoroutines should not have any logic associated to reset signals. Instead, when appropriate, the parent should propagate its own cancellation to its subcoroutines if the parent itself becomes reset.

Because of this, the issue that coroutine reset behavior cannot be provided by default in a design that unifies the declaration of a coroutine object and its associated execution does not apply to subcoroutines.

Additional changes to the proposed design can be made to further simplify the use of subcoroutines:

- Subcoroutine objects can be extended with output parameters, and the parent may await upon the completion of a subcoroutine, at which point it receives the output parameters are returned.

  As the subcoroutine may complete execution before the parent is ready to await upon it, this makes it necessary to preserve the output parameters. This has the downside that the output parameters would be required to be serializable.

- Scoped canceling message handling makes use of subcoroutines in order to emulate `async` method calls as triggers (see Section 4.4.2). These have the specific usage pattern of starting a subcoroutine and then immediately waiting for its completion; canceling the subcoroutine if the wait is canceled for any reason. It's possible to simplify this usage pattern by representing it through a utility `async` submethod of subcoroutine objects – `run()` – with built-in support from the compiler to allow its call to be used as a trigger within a `race`/`concurrently`.

`try-except async` can be used for idiomatic cancellation handling within a subcoroutine (see Section 4.5).

The syntax for declaring a subcoroutine object is analogous to methods:

```
subcoroutine name(input_1_t input_1, ...)
          -> (output_1_t out_1, ...) {
  ... // Statements, in asynchronous scope. Must return the output
  ↪  parameters.
}
```

Subcoroutine objects have the following submethods:

- `start()` and `restart()`. These methods are used in order to start the subcoroutine, and must be provided arguments according to the required input parameters declared with the subcoroutine. If the subcoroutine is already in progress, `start()` logs an error message, while `restart()` forces the subcoroutine to restart execution.

- `cancel()` – cancels the subcoroutine if running.

- `run()` – described above

Declared subcoroutines may also be `await`-ed upon, like channels. This causes the coroutine to suspend execution until the subcoroutine has run and terminated successfully with output parameters – which are then returned by the `await` expression.

4.3.2 Unbounded Design

Lifting the bounded state restriction introduces the possibility of *dynamically* creating coroutine instances as asynchronous code is executed. This could be leveraged in order to address both issues of the subcoroutine pattern – subcoroutines would no longer need to be declared separately from their usage point, and such `async` methods would be safe to share between multiple coroutines, as each individual call of the `async` method dynamically creates its own set of coroutine instances.

Assuming the semantics of duplicate listening is precisely defined, implementation of `async` method calls as triggers within a `race`/`concurrently` statement becomes unproblematic, by representing such triggers with dynamically created subcoroutines, linked to the parent coroutine in the sense that

- If the associated participant exits the `race`/`concurrently` for any reason, the dynamically created coroutine becomes terminated, and resources for it are deallocated.
- If both the subcoroutine and its parent listens to the same channel, then the subcoroutine is resumed *before* its parent if a message is sent through that channel. This guarantees that any subcoroutine will be able to react to a message before its parent may cancel that subcoroutine by its own reaction to that message.

`async` method calls as triggers alone aren't sufficient, as it doesn't allow for a coroutine to establish and maintain dynamically created subcoroutines to execute concurrently with its own code without extreme amounts of boilerplate. Due to this, the unbounded design is also extended such that dynamically created coroutines can also be explicitly created and waited upon at a later time after their creation. In order for to guarantee such dynamically created coroutines do not accidentally lead to resource leaks – and in order to simplify implementation – any dynamic coroutine instances must apply over a scope – and if the scope is left for any reason, the dynamic coroutine is terminated and its resources released. The specification for such subcoroutines is as follows:

- A new non-compound object type, `subcoroutine`, is introduced. Such objects are *dynamic*, and may only be declared locally inside of `async` methods.
- `subcoroutine` objects are declared as follows:

  `subcoroutine` *`name`*`:` *`awaitexpression`*`;`

  Where *name* is the identifier for the subcoroutine, and *awaitexpression* is any `await` expression.

- The statement that declares the subcoroutine will cause a subcoroutine to be dynamically created and execute the parameterized await expression.
- Any subcoroutine can be `await`-ed upon, which suspends execution of the calling parent coroutine until the subcoroutine has finished execution, if it hasn't already, and returns the result of evaluating the `await` expression. As this requires the result to be stored, it must be serializable – in particular, if the `await` expression is an `async` method call, all output parameters of the `async` method are required to be serializable.
- All resources for the subcoroutine are released once the parent coroutine leaves the scope of the declared subcoroutine for any reason. If the subcoroutine has not finished execution by that point, it is canceled.

  This guarantees that all resources necessary to create subcoroutines are eventually released, as long as the parent coroutine eventually leaves the scope.
- Dynamically created subcoroutines can be canceled preemptively by their parent through the built-in `cancel()` submethod. Subcoroutines also offer the built-in `completed()` and `canceled()` submethods to check if they have been completed, or have been canceled, respectively.

  `await`-ing upon a canceled subcoroutine is *forbidden*; suggested behavior is that a high-severity error is logged, and the parent coroutine becomes permanently suspended on that `await` unless externally canceled.

A notable use case for such subcoroutines would be for non-canceling message handling, as follows:

```
saved local output;
{
  // subroutine() is an async method that returns an uint8
  subcoroutine subcor: await subroutine();

  while (!subcor.completed()) {
    race {
      case (output = await subcor);

      case (await chanA) {
        respond_to_msg_A();
      }

      case (await chanB) {
        respond_to_msg_B();
      }
    }
```

```
  }
}
// subcoroutine output now available in output variable
```

4.4 Scoped Message Handling

The basic design only has one mechanism which asynchronous code may leverage for on-site simultaneous handling of different messages: the `race` and `concurrently` statements. In particular, the basic design does not provide any intrinsic support for *scoped message handling* – attaching message handlers to entire blocks of asynchronous code, such that any attached message handler can be triggered and executed at any suspension point within the block.

Scoped message handling may take on two main forms – *canceling* and *non-canceling*. Canceling scoped message handling *cancels* the execution of the scoped block if any attached message handler is triggered. This is expected to be the most common form of scoped message handling, as it can be used to allow and/or handle *external cancellation* of an asynchronous subroutine executed by a coroutine. For example, this could be leveraged in order to permit on-site recovery of resets within asynchronous code, or to cancel an asynchronous subroutine following a timeout or external message.

The naive alternative to scoped message handling is to manually attach each handler to every suspension point within the asynchronous block via `race` and `concurrently` statements, but this has three significant issues:

(a) It requires significant amounts of non-idiomatic boilerplate; *every* `await` expression must be transformed into a `race` statement, and this could incur the need for further changes in the code.

(b) By having it be the responsibility of the subroutine to attach handlers unimportant for its own logic, separation of concerns is violated. This is not only intrusive when programming the subroutine, but also makes the subroutine inflexible to use, and difficult to extend upon. In order to for a coroutine to make use of a subroutine at any point of its logic, the subroutine must be tailored to support that usage point.

For example, consider if the subroutine is used at two different points within the logic of the parent coroutine – however, each point has *different sets* of message handlers that need to be attached to the subroutine. This means that the subroutine needs to be programmed such that it may support either set of message handlers, and must have some means of knowing which set should be used (e.g. via a flag).

This is especially problematic during active development; anything that would affect the possible sets of message handlers that may be attached to the subroutine – such as new usage points, or the addition or removal of new message handlers – would require extensive refactoring to be supported.

(c) It's impossible to attach message handlers to suspension points within asynchronous code not defined by the user. This means that if the subroutine contains a call to a foreign `async` method or is itself a foreign `async` method, then this solution can't be applied.

4.4.1 Example Problem

The code below demonstrates an incomplete program for a coroutine which – as part of its logic – calls a foreign `subroutine() async` method. The program needs to be modified such that a message handler for `cancel`, as well as a timeout – is attached to the `subroutine()` call in `main.coroutine()`.

```
bank regs {

  register result_valid size 1 @0x0 is read_only;

  register running is (write) @0x1 {
    method write(uint64 v) {
      if (get() > v) { // 1 --> 0; cancel read request
        main.cancel.send();
      } else if (v > get()) { // 0 --> 1; start read request
        main.start.send();
      }
    }
  }

  register result size 1 @0x2 is read_only;
}

coroutine main is (hard_reset) {
  param timeout = 2;
  channel () start;
  channel () cancel;

  async method coroutine() {
    while (true) {
      await start;
      regs.result_valid.set(0);
      local uint8 out = await subroutine();
      regs.result.set(out);
      regs.result_valid.set(1);
      regs.running.set(0);
    }
  }
}

// Body unknown; cannot be modified.
async method subroutine() -> (uint8);
```

4.4.2 Existing Solutions

Attaching message handlers to a subroutine of a coroutine can be accomplished by representing that subroutine through a subcoroutine – which the parent coroutine communicates with through channels in order to start the subroutine and wait for its completion. This means `race` and `concurrently` statements may be used to listen for other messages while awaiting the completion of the subroutine – optionally canceling the subroutine when another message is received.

By representing the subroutine call through a subcoroutine `sub`, this can be done in the basic design by modifying the body of `main` to the following:

```
coroutine main {
  param timeout = 2;
  channel () start;
  channel () cancel;

  async method coroutine() {
    while (true) {
      await start;
      regs.result_valid.set(0);
      subroutine.start.send();
      race {
        case (local uint8 out = await sub.done) {
          regs.result_valid.set(1);
          regs.result.set(out);
        }
        case (await delay timeout s) {
          log info: "subroutine() call timed out";
          // Cancel execution of the subcoroutine
          sub.restart();
        }
        case (await cancel) {
          log info: "main canceled";
          sub.restart();
        }
      }
      regs.running.set(0);
    }
  }

  coroutine sub {
    channel () start;
    channel uint8 done;
    async method coroutine() {
      while (true) {
        await start;
        local uint8 out = await subroutine();
        after: done.send(out);
```

```
      }
    }
  }
}
```

Although this is a valid solution to the problem, it has two issues related to the use of static subcoroutine instances:

- Any `async` method that makes use of subcoroutines in order to perform scoped message handling can't be safely shared between multiple coroutines.
- The declaration of static subcoroutine instances are boilerplate intensive.

Because of this, each of the iterated designs must either receive an extension designed specifically for scoped message handling, or receive extensions that allow subcoroutines to be idiomatically used for scoped message handling.

4.4.3 Extending the Bounded Design

Section 4.3.1 extends the bounded design such that the boilerplate associated with statically declared subcoroutine instances are greatly reduced, and thus solves the primary issue of scoped message handling. In particular, the `run() async` submethod of subcoroutines under the bounded design receives built-in support that allows it to be used as a trigger in a `race/concurrently` statement – making it suitable for canceling scoped message handling.

This allows the problem example to be solved as follows:

```
coroutine main {
  param timeout = 2;
  channel () start;
  channel () cancel;

  async method coroutine() {
    await start;
    regs.result_valid.set(0);
    race {
      case (local uint8 out = await sub.run()) {
        regs.result_valid.set(1);
        regs.result.set(out);
      }
      case (await delay timeout s) {
        log info: "subroutine() call timed out";
      }
      case (await cancel) {
        log info: "main canceled";
      }
    }
```

```
    regs.running.set(0);
  }

  subcoroutine sub() -> (uint8) {
    return await subroutine();
  }
}
```

As such, the only unresolved question is if it is possible to extend the bounded design with a feature for scoped message handling that would eliminate the secondary issue of subcoroutines – methods making use of subcoroutines can't be safely shared between multiple coroutines, but need to be duplicated.

Any minimal extension which would allow for scoped message handling – canceling or not – such that any `async` method performing it is safe to share between multiple coroutines would have the following issues:

- It would enable *duplicate listening*; a scope may contain a listen to a channel which is also listened to by an attached message handler.
- The maximum number of simultaneous listens that a coroutine may perform at any given suspension point is no longer be determinable by the specific suspension point alone. This means that the scheme for the efficient static allocation of channel queue nodes as described in Section 3.2.4 could no longer be applied.

Theoretically, duplicate listening could be prevented by having the DML compiler keep track of what channels a coroutine may listen to within a particular scope (including any listens within any `async` method calls), and reject the attachment of any message handler to any scope that may listen to same channel as the message handler. However, this would be complicated to implement – possibly more complicated than it would be to support duplicate listening – and presents severe usability issues, as it can be difficult for a user to determine what channels any given `async` method call may listen to, and how to resolve the issue if the attachment is rejected.

Assuming duplicate listening were disallowed,[8] the second issue could theoretically be resolved by modifying the scheme to also analyze every scope to which message handlers are attached in order to determine the maximum number of channels that a coroutine may listen to simultaneously. This would significantly increase implementation complexity.

In conclusion, the complexity of any alternative to the subcoroutine approach are unlikely to justify potential benefits; the duplication issue of static subcoroutines presents only a minor issue which can easily be resolved by the user. As such, no further extensions to the bounded design are made in order to support scoped message handling.

[8] It may be still be possible to apply this scheme if duplicate listens were supported, depending on how that support is implemented.

4.4.4 Extending the Unbounded Design

Section 4.3.2 extends the unbounded design with the feature to dynamically create subcoroutines. In particular, it permits the use of `async` method calls as triggers, providing a natural means for scoped canceling message handling. This allows the problem example to be solved as follows:

```
coroutine main {
  param timeout = 2;
  channel () start;
  channel () cancel;

  async method coroutine() {
    await start;
    regs.result_valid.set(0);
    race {
      case (local uint8 out = await subroutine()) {
        regs.result_valid.set(1);
        regs.result.set(out);
      }
      case (await delay timeout s) {
        log info: "subroutine() call timed out";
      }
      case (await cancel) {
        log info: "main canceled";
      }
    }
    regs.running.set(0);
  }
}
```

As subcoroutine instances are dynamically created, methods making use of them can be safely shared between multiple coroutines, eliminating both issues with the static subcoroutine approach to scoped message handling.

4.5 Cancellation Propagation and Handling

It's possible for the execution of asynchronous code to be canceled externally – the execution of a coroutine can be terminated by a call to `restart()`; and in the unbounded design, the execution of an `async` method call as a trigger can be canceled by a parent `race` being completed.

This presents an issue if any resources are acquired during the execution of asynchronous code – as these will not be released if the coroutine is externally canceled. There is therefore a need for asynchronous code to be able to handle cancellation.

4.5.1 Existing Solutions

As stated in Section 4.4, scoped message handling via subcoroutines can be used in order to handle cancellation. Any subcoroutine that needs to be able to perform on-site cancellation handling may do so by having an associated `channel` object that represents cancellation, to which a message is sent before the subcoroutine is restarted proper. This can be followed by a `throw` or a dummy suspension in order to idiomatically resume cancellation. Such on-site cancellation handling can be used to propagate cancellation to child subcoroutines of the canceled subcoroutine.

This requires the parent coroutine calling a wrapper function around `restart()` – which performs a send to the cancellation channel – rather than calling `restart()` directly.

For example, the following demonstrates how – under the basic design – a library may declare a template to instantiate a particular cancellable subcoroutine for some process that, in turn, uses a cancellable subcoroutine of its own. The execution of a cancellable subcoroutine can be canceled by its parent by calling the `cancel()` submethod.

```
template a_cancellable_subcoroutine is coroutine {
  channel () start;
  channel uint8 done;
  channel () canceled;
  channel () never;

  coroutine sub
    is another_cancellable_subcoroutine;

  async method coroutine() {
    while (true) {
      await start;
      local output_t output = await main();
      done.send(output);
    }
  }

  async method main() -> (uint8) {
    ...

    // Here, some form of protected resource
    // is acquired, and needs to be released
    // if the subcoroutine is canceled.
    get_resource();

    // The child subcoroutine is started
    sub.start.send(some_param);

    race {
```

```
      // Normal line of execution
      case (local bool result = await sub.done) {
        ...
      }

      case (await canceled) {
        // Propagate cancellation to the started child subcoroutine
        sub.cancel();

        release_resource();

        // Return execution to the caller of cancel() by
        // suspending execution on a termination channel
        await never;
      }
    }
    release_resource();

    ...
  }

  method cancel() {
    // Perform on-site cancellation handling
    canceled.send();

    // Cancel the subcoroutine proper
    restart();
  }
}
```

4.5.2 Extending the Bounded and Unbounded Design

Although the extensions to the bounded and unbounded designs described in Section 4.3 simplify the use of subcoroutines, they do not address the boilerplate needed to allow for on-site cancellation handling – in particular:

- The need to create a corresponding cancellation channel
- The need to create a `cancel()` wrapper around `restart()`/`terminate()`
- Cancellation propagation to any subcoroutines must be done explicitly.

In addition, the reliance on a cancellation channel negates the usefulness of dynamically created subcoroutines that the unbounded design supports, as the cancellation channel for any subcoroutine must be statically declared. This approach also can't be used for cancellation handling inside of `async` method calls that are used as triggers.

Due to the above, there exists a need to provide built-in support for cancellation handling. To this end, both the bounded and unbounded designs

are extended by introducing the `try-except async` statement, which can be only used in asynchronous scopes, and is invoked as follows:

```
try main except async handler
```

where both *main* and *handler* are (compound) statements.

Executing a `try-except async` statement is the same as executing *main*, except if execution would be externally canceled during the execution of *main* for any reason, then *handler* is executed before the cancellation is resolved.

If external cancellation occurs at a suspension point enclosed by multiple `try`-except async statements, then each handler is executed in the order of nearest enclosing `try-except async` statement.

Both *main* and *handler* inherit the scope from before the beginning of the `try ... except async ...` statement, with the exception that *handler* is considered to be in synchronous scope, and any previously declared `local` variables and method parameters are not in scope inside *handler*. In addition, *handler* may not make use of `return` or `throw` statements, and any method call to a `throws` method within *handler* must be enclosed in a `try ... except ...` statement, regardless if the parent method is declared `throws`.

4.6 Usability Issues of Synchronous Sends

A property of the basic design is that the `send()` submethod of channels is *synchronous* – all coroutines listening to the channel are the message and consequently resumed before `send()` returns. However, this behavior has a number of issues:

- As shown in Section 3.6, a coroutine may inadvertently initiate a request to which a response provided *synchronously*, thus not allowing the coroutine to listen to the response in time. This is expected to be a common source of bugs.
- Synchronous sends have a number of problematic interactions with other elements of the basic design which can't be satisfactorily resolved – and are thus considered *forbidden*, even though this can't be statically enforced. These are explained in Section 3.7.

The *immediate* `after` statement was developed in order to address these issues. Any `send()` delayed via an immediate `after` statement is executed only once control would be returned back to the simulation engine – and at that point, all coroutines of the DML program are guaranteed to either be suspended or terminated, thus guaranteeing that the call is unproblematic. Such delayed sends are called *asynchronous*.

Due to the usability issues presented by synchronous sends, asynchronous sends are expected to present the most common use of `send()` by far. This

raises the question if `send()` should be modified such that it is always asynchronous.

Although synchronous sends are not expected to be common, these satisfy certain needs that asynchronous sends do not:

- Performing a synchronous send allows for the sender to know the number of coroutines that accept that send. This is not possible with asynchronous sends.
- Synchronous sends are necessary for synchronous communication with a coroutine from synchronous code. This can be leveraged in order to have output parameters associated with a channel via pointers, as described in Section 3.2. Synchronous sends are also useful when it is necessary to guarantee that all coroutines affected by the send resume execution before a different action is taken; for example, Section 4.5.1 shows how this can be leveraged in order to allow a subcoroutine to perform on-site cancellation handling before it is canceled proper.
- The current stack is guaranteed to be valid throughout a synchronous send, making it safe to pass pointers to local variables via a send – unlike with asynchronous sends.
- Synchronous sends are more desirable in any context where they are known to be unproblematic – for example, as part of a method call whose execution is already delayed via the immediate `after` statement.

  This allows for wrapping logic around a synchronous send with all the previously mentioned benefits, and then making that use safe by delaying the entire block through immediate `after`.

Because of this, the ability to perform synchronous sends should not be removed in its entirety; but it is clear that it should not be the default behavior.

Therefore, both the bounded and unbounded designs are modified such that the `send()` submethod of channels performs an *asynchronous send*. The variant of `send()` featured in the basic design – which performs synchronous sends – is now instead offered through the newly introduced `send_now()` channel submethod.

4.7 Preservation of Local Variables

In the basic design, declared method parameters and `local` variables can't be used past any `await` expression or `race`/`concurrently` statement. If this wasn't enforced, then such variables would need to be stored and serialized, which is problematic for two reasons:

- As each coroutine needs to preserve stack-allocated variables independently, and only a small subset of all possible stack-allocated variables are in use at any given point of the simulation for each coroutine, allocating storage for each possible usage of stack-allocated variables would violate the bounded state restriction.
- The contents of stack-allocated variables can't be serialized in general; for example, pointers can't be serialized.

However, the first issue only applies while the bounded state restriction is enforced, and the second issue can be addressed by introducing separate variable kinds for preserved respectively non-preserved stack-allocated variables. This means that this feature can be supported by the unbounded design.

The following extensions is introduced to the unbounded design:

(a) A new variable kind is introduced – `saved local` variables. These may only be declared inside of asynchronous scopes, and are done so analogously to `local` variables.

(b) Unlike `saved` variables, `saved local` variables may be initialized with non-constant expressions. `saved local` variables are also not shared between calls to the same `async` method.

(c) Unlike `local` variables, and like `saved` variables, the type of `saved local` variables must be *serializable*, and this is enforced at compile-time.

(d) Unlike `local` variables, the lifetimes of `saved local` variables are not affected by possible suspension points – they are not subject to the scoping rules outlined in Section 3.4.3 and Section 3.5.2.

(e) `async` method parameters may be declared as `saved local` through a `saved` prefix, as follows:

```
async method foo(saved uint8 bar, saved bool baz, ...) {
  ...
}
```

A possible modification to this design is to have preserved local variables be the default, and relegate local variables with limited lifespans to a separate variable kind – e.g. called `temp`. The benefit of this is that the scoping behavior of method parameters and `local` variables would be completely analogous to regular methods; however, such variables would still differ in that the parameterized type must be serializable – and unlike `saved local` variables, that restriction isn't communicated through the name alone.

4.8 Bodies Associated with non-`await` Participants

The current design of `race`/`concurrently` statements only allow for bodies associated to `await` participants, allowing for logic to be attached with each individual trigger of a `race`/`concurrently` statement. However, it may also be desirable to attach logic to the successful completion of `race`/`concurrently` participants as a whole – which is thus executed when any immediate participant has completed, or every immediate participant has completed, respectively. The fact that the basic design does not support this presents two particular feature gaps:

- Attaching logic to the completion of a `race` would allow for specifying common code to be executed after any participant completes – for example, this could be used to release any resources needed for the other participants.
- Attaching logic to the completion of a `concurrently` would allow a user to execute logic that is only valid once every participant is known to be completed – for example, this allows for a coroutine to wait for two parameters to be passed to it through `channels`, and execute a body making use of those parameters once received.

To an extent, it's possible to compensate for these gaps:

- The feature is not needed for `race` or `concurrently` statements with only `await` participants; any logic that needs to be executed following the completion of the `race`/`concurrently` can be implemented simply by executing it after the `race`/`concurrently` statement.
- Any logic associated with the completion of a `race` participant with only `await` immediate participants can be implemented by executing it at the end of the bodies of each `await` participant. To avoid code duplication, it may become necessary to separate that logic to a method call.
- If logic needs to be associated with the completion of a single `concurrently` that can be canceled by a timeout or external cancellation, then executing that logic after the related `race` statement can be controlled by a simple flag that indicates whether or not cancellation occurred.

However, certain usage patterns are impossible to idiomatically implement without this feature, which thus presents usability issues of the designs.

4.8.1 Problem Example

As a motivating example of why the feature is necessary, consider a `concurrently` statement with two `concurrently` participants, each of which should have

associated logic to be executed at the moment that participant completes. As the `concurrently` statement isn't left until both participants complete, this isn't possible to implement through statements after the `concurrently` statement.

To make the example more concrete, the following scenario is presented:

- One set of listens is to receive a *packet*, as well as a *target device address* to propagate the packet to. Once both have been received, the packet should be propagated.

  The associated channels are `channel uint8 get_packet` and `channel uint64 get_propagation_target`.

- One set of listens is to receive a *command*, as well as to receive *permission* to execute that command. Once both have been acquired, the command should be executed.

  The associated channels are `channel () get_permission` and `channel command_t get_command`, where `command_t` is a serializable but otherwise abstract datatype.

The following code represents this usage scenario:

```
async method propagate_packet_and_execute_command() {
  saved uint8 packet;
  saved uint64 target;
  saved command_t command;

  concurrently {
    // Once completed, "propagate_packet(packet,target)"
    // should be executed.
    concurrently {
      case (packet = await get_packet);
      case (target = await get_propagation_target);
    }

    // Once completed, "execute_command(command)"
    // should be executed.
    concurrently {
      case (await get_permission);
      case (command = await get_command);
    }
  }
}
```

4.8.2 Basic Design

The most idiomatic approach in the basic design is to represent each `concurrently` participant through a subcoroutine as follows:

```
coroutine propagate_packet_cor {
  channel () start;
  channel () done;

  async method coroutine() {
    while (true) {
      await start;

      saved uint8 packet;
      saved uint64 target;
      concurrently {
        case (packet = await get_packet);
        case (target = await get_propagation_target);
      }
      propagate_packet(packet,target);

      done.send();
    }
}

coroutine execute_command_cor {
  channel () start;
  channel () done;

  async method coroutine() {
    while (true) {
      await start;

      saved command_t command;
      concurrently {
        case (await get_permission);
        case (command = await get_command);
      }
      execute_command(packet,target);

      done.send();
    }
}

async method propagate_packet_and_execute_command() {
  propagate_packet_cor.start.send();
  execute_command_cor.start.send();
  concurrently {
    case (await propagate_packet_cor.done);
    case (await execute_command_cor.done);
  }
}
```

This solution is deemed too boilerplate heavy to be acceptable; the complex-

ity only grows if more participants are introduced, or if the `concurrently` participants could become canceled – in which case the subcoroutines need to be modified to support cancellation.

4.8.3 Extending the Bounded and Unbounded Designs

In order to allow for a body to be associated with `race`/`concurrently` statements/participants, both the bounded and unbounded designs are extended with a new feature – `completed` declarations. The body of any `race`/`concurrently` may feature a `completed` declaration together with the declarations of its immediate participants. A `completed` declaration specifies statements to be executed after the `race`/`concurrently` it belongs to becomes completed.

Specification A `completed` declaration is given as follows:

`completed` *`body`*

where *body* must be a statement. As `completed` declarations are not participants, `completed` may not be prefaced with a guard, and does influence the check for whether the parent `race`/`concurrently` should be considered completed. A maximum of one `completed` declaration is allowed per `race`/`concurrently` participant (not including any `completed` declaration of any child `race`/`concurrently` participant).

Once a `race`/`concurrently` statement/participant is considered completed (*without* having been canceled), then the body of its `completed` declaration will be executed *after* the bodies of any triggered participants, and *before* the coroutine re-suspends execution at the `race`/`concurrently` statement – or, if the statement as a whole is considered completed, before execution resumes past the statement.

If the bodies of multiple `completed` declarations are to be executed at the same point in time, then these are executed in declaration order, *but prioritizing children*. That is to say, all pending `completed` executions are executed top-to-bottom, depth-first recursively, with the exception that any pending `completed` execution of a `race`/`concurrently` is only executed once each pending `completed` of the child participants have been executed.

The body of a `completed` declaration is in asynchronous scope only if the `race`/`concurrently` it belongs to is *not* an immediate or indirect descendant of a `concurrently` statement or participant. Otherwise, the body is in synchronous scope.[9]

[9]Note that the `completed` declaration may belong to a `concurrently` statement/participant without it affecting if the body of the `completed` declaration is in asynchronous scope.

With this extension, the problem scenario can be implemented in the bounded design as follows (assuming the parent method is only executed by one coroutine at a time):

```
saved uint8 packet;
saved uint64 target;
saved command_t command;

concurrently {
  concurrently {
    case (packet = await get_packet);
    case (target = await get_propagation_target);

    completed {
      propagate_packet(packet, target);
    }
  }

  concurrently {
    case (await get_permission);
    case (command = await get_command);

    completed {
      execute_command(command);
    }
  }
}
```

In the unbounded design, the above can be modified to use `saved local` variables, instead.

An identified issue with this solution is that the need to separately declare `saved` or `saved local` associated with a particular `race`/`concurrently` is unidiomatic. Because of this, `race`/`concurrently` statements/participants are extended such that variables of any kind but `local` may be declared within these; such declarations are equivalent to declaring these variables separately within the parent method, except that they are only accessible within the scope of the `race`/`concurrently` in which they are declared. This allows the above to be rewritten into the following:

```
concurrently {
  concurrently {
    saved uint8 packet;
    saved uint64 target;

    case (packet = await get_packet);
    case (target = await get_propagation_target);

    completed {
      propagate_packet(packet, target);
```

```
    }
  }

  concurrently {
    saved command_t command;

    case (await get_permission);
    case (command = await get_command);
    completed {
      execute_command(command);
    }
  }
}
```

5 Evaluation

In order to evaluate the developed designs, multiple existing DML modules implementing asynchronous logic through state machines were rewritten to instead make use of coroutines under the various designs. The original modules and their rewritten variants are used in order to identify particular usage patterns related to the development of state machines for asynchronous logic, how common each particular usage pattern is, and how well each design addresses such usage patterns. This is used to evaluate what particular features of each design have proven to be of practical worth, as well as usability issues that were not addressed during development.

As the impact on code of the various designs cannot be quantitatively measured in a well-defined way, the evaluation will be done by comparing code size and discussing the ease of use of each design. Code size serves as a simple indicator of the impact of boilerplate code. Even so, it is imperfect as a metric – it can be skewed by multiple factors unrelated to the coroutine design, such as differences in coding style.

The DML modules studied originated from confidential codebases internal to Intel, and so will not be presented in their original form within this report. Any code included in this section corresponds to observed patterns in the original modules, or even particular excerpts, but will be presented such that the code does not reveal any confidential information.

5.1 Overview

In total, 28 finite state machines (FSMs) were rewritten to have their operation driven by coroutines. Although each FSM typically corresponded to a single dedicated DML module, code interacting with an FSM often spanned multiple modules, and these were rewritten as needed.

Three design forks were evaluated: the two primary bounded and unbounded designs, and a variant of the bounded design featuring non-compound

coroutine objects as described in Section 4.2.2. Boilerplate code incurred by the use of FSMs is roughly proportional to the number of associated states, and so the average reduction of lines of code per state is used to evaluate the success of each coroutine design. The total number of states across all of the original FSMs roughly amount to 277.[10]

The cumulative number of removed lines and newly inserted lines were measured for each design, from which the total number of lines reduced and lines reduced per state may be derived. The results are shown in Table 1.

In addition to these results, the impact of each coroutine design on individual usage patterns featured in the original FSM modules is studied and analyzed in Section 5.2. The majority of discussion, code excerpts, and metrics presented in that section only concerns the primary bounded design, as the practical differences between the designs are minimal for the vast majority of rewritten code. That is reflected by the similar results for each design shown in Table 1. Individual designs are only discussed when their differences are relevant.

The number of reduced lines for each design shown in Table 1 is significantly inflated by a small number of large DML modules featuring unique patterns that result in dramatically reduced code size when rewritten. The nature of these usage patterns are detailed in Appendix C. As these do not accurately reflect the usage patterns of the other studied FSMs, a better indication of the performance of each design is given by performing the above calculations on a restricted subset of FSMs without these outliers. This was done on a subset of 26 FSMs, with a total of 252 states. The results are shown in Table 2.

Design Fork	Lines Removed	Lines Inserted	Lines Reduced	Lines Reduced per State
Bounded, Compound	6366	3705	2661	9.60
Unbounded, Compound	6357	3707	2650	9.57
Bounded, Non-compound	6369	3691	2678	9.67

Table 1: Code reduction results for each evaluated design across all 28 FSMs

[10]This count is not exact; FSMs may feature states that are unused.

Design Fork	Lines Removed	Lines Inserted	Lines Reduced	Lines Reduced per State
Bounded, Compound	5364	3422	1942	7.70
Unbounded, Compound	5356	3425	1931	7.67
Bounded, Non-compound	5368	3409	1959	7.77

Table 2: Code reduction results for each evaluated design across a subset of 26 FSMs without outliers

5.2 Usage Patterns

5.2.1 Linear Asynchronous Logic

All coroutine designs target the development of mostly linear asynchronous logic; such logic corresponds to FSMs where most states only have a single possible state transition. In order to properly evaluate the designs for this expected typical usage pattern, the same measurements as in Section 5.1 were applied to a restricted subset of 11 FSMs that are predominantly linear in nature, and do not include any usage patterns that are dramatically reduced in size when rewritten. For this subset, 817 lines of code were reduced across 114 states – on average, 7.17 lines of code reduced per state.

This result can be better reinforced by observing how such code is typically affected by the use of coroutines. Although the specific implementation and associated boilerplate of FSMs varies, the most commonly observed implementation of FSMs (17 out of the total 28) have their core logic implemented through a central reaction and progression method, intended to be called by external code in order to notify the FSM, causing it to act and progress.

Such FSMs represent their states as constants, and make use of a `switch` statement to act according to the current state. Any actions and transitions of the FSM are guarded by checks to ensure that the context for the notification is correct. The most common form of notification context are through different *event kinds*, of which must be explicitly provided when calling the notification method – 19 of all 28 FSMs feature explicit events as notification context. The following is a simplified but largely accurate example of the structure of such FSMs:

```
param EVENT_A = 0;
param EVENT_B = 1;
...
```

```
param STATE_A = 0;
param STATE_B = 1;
...

saved uint64 curr_state = STATE_A;

// Notification method, containing the core FSM logic.
// Notification must be coupled with info about event kind.
method somefsm(uint64 event) {
  switch(curr_state) {
  case STATE_A:
    if (event == EVENT_A) { // Context check
      ... // react to EVENT_A
      curr_state = STATE_B;
    }
    break;
  ...
  }
}
```

With this structure, every simple linear state – states that only accept one particular notification context (here, one particular event kind) and only transitions to exactly one different state – incurs at least 6 lines of boilerplate.

- The declaration of the constant labeling the state (1 line)
- The `case` statement of the switch in `somefsm()`, and associated `break` (2 lines)
- The check for correct notification context (2 lines or more)
- The specification of which state to transition to. (1 line)

In comparison, the following demonstrates how the interface of such an FSM (the key property being the notification method) can be preserved while utilizing coroutines:

```
param EVENT_A = 0;
param EVENT_B = 1;
...

coroutine somefsm_coroutine {
  channel uint64 event_chan;

  async method coroutine() {
    session uint64 event;

    // Corresponds to STATE_A
    event = await event_chan where (event == EVENT_A);
    ... // react to EVENT_A
```

```
    // Corresponds to STATE_B
    event = await event_chan where (event == EVENT_B);
    ... // react to EVENT_B
  }
}

method somefsm(uint64 event) {
  somefsm_coroutine.event_chan.send_now(event);
}
```

This showcases how conditional listens can be utilized in order to guard progression, allowing simple linear states to be translated into as little as one line of boilerplate through the following pattern:

```
  await notification_chan where (context_check());
```

Thus, for this form of FSMs, coroutines can be expected to reduce code size by least 5 lines of code per state.[11]

The observed average reduction of 7.17 lines of code per state for predominantly linear asynchronous logic is plausible when taking into account other miscellaneous causes of reduction in code size, most notably reduction in boilerplate related to logging (see Section 5.2.2).

Another reason is that the above reflects the most common implementation of FSMs, while 6 of the 28 rewritten FSMs instead relied on the use of a DML template – called `fsm` – offered by a library within the one of the studied codebases. This template allows the user to represent events and states through `group` declarations. The following demonstrates how an FSM may be declared through the use of this template:

```
bank somefsm is fsm {
  group events {
    group event_A;
    group event_B;
    ...
  }

  group state_A is fsm_init_state {
    group event_A is fsm_event_handler {
      method handle() {
        .. // react to event_A -- not boilerplate
        state_B.set_fsm_state();
      }
```

[11] Not all rewritten FSMs make use of this pattern. In fact, for the specific form of FSMs where notification context consists of an event kind – as represented by a constant – the rewritten modules leverages different approaches in order to eliminate the verbose syntax of conditional listens. However, no matter the approach, the reduction in code size is always similar as what is demonstrated here – at least 5 lines of code for simple linear states.

```
    }
  }

  group state_B is fsm_state {
    group event_B is fsm_event_handler {
      method handle() {
        ... // react to event_B -- not boilerplate
        state_C.set_fsm_state();
      }
    }
  }

  ...
}
```

Using the same analysis as above, simple linear states under this representation incur 7 lines of boilerplate per newly introduced state (not including empty lines). The benefits of using the `fsm` template lie in the high level interface offered by the template, including automatic logging of state transitions and received events.

5.2.2 Logging

One particular usage pattern observed across almost every FSM studied was the logging of any event or state change related to the operation of the FSM. This need was never addressed during the development of the coroutine designs, and presents the largest oversight during development that was discovered during evaluation.

In particular, 27 of the 28 FSMs featured logging for one or more of the following:

- Any state transition, logging the previous state and the next state.

  This corresponds to a coroutine progressing from one suspension point to another.

- When applicable, any event the FSM becomes notified of.

  This corresponds to any message which is propagated to a coroutine via a channel it is listening to.

- When applicable, when the FSM is not able to handle a particular event – that is, it receives a particular event at a state when the event is not expected.

  This correspond to any message sent via a channel which is not accepted by any coroutine listening to the channel – or if no coroutine is listening to the channel.

Of these three, unhandled events are especially important to log, as these typically represent errors which may impede the proper operation of the FSM, and are logged as such. In contrast, state transitions and received events are logged at low severity, which implies such logging is typically only leveraged for additional introspection into the operation of the FSM while debugging.

The boilerplate associated to logging varies between FSM implementations. The low-level `switch`-based implementation presented in Section 5.2.1 is a simplification mainly due to the absence of logging logic: in addition to the general structure presented there, variations of the following auxiliary declarations and methods for logging are also typically present:

```
loggroup somefsm;

method stringify_state(uint64 state) -> (const char*) {
  switch (state) {
  case STATE_A: return "STATE_A";
  case STATE_B: return "STATE_B";
  ...
  default: return "Unknown state";
  }
}

method stringify_event(uint64 event) -> (const char*) {
  switch (event) {
  case EVENT_A: return "EVENT_A";
  case EVENT_B: return "EVENT_B";
  ...
  default: return "Unknown event";
  }
}

method log_received_event(uint64 event) {
  log info, 4, somefsm: "Received event %s in state %s"
                      , stringify_event(event)
                      , stringify_state(curr_state);
}

method log_unhandled_event(uint64 event) {
  log error, 2, somefsm: "Unhandled event %s in state %s"
                       , stringify_event(event)
                       , stringify_state(curr_state);
}

method set_somefsm_state(uint64 new_state) {
  if (new_state != curr_state) {
    log info, 4, somefsm: "State transition: %s -> %s"
                        , stringify_state(curr_state)
                        , stringify_state(new_state)
```

```
  }
  curr_state = new_state;
}
```

These are then leveraged in the central transition method as follows:

```
method somefsm(uint64 event) {
  log_received_event(event);
  local bool event_handled = false;
  switch(curr_state) {
  case STATE_A:
    if (event == EVENT_A) { // Context check
      event_handled = true;
      ... // react to EVENT_A
      set_somefsm_state(STATE_B);
    }
    break;
  ...
  }
  if (!event_handled) {
    log_unhandled_event(event);
  }
}
```

Note that now every newly introduced state introduces two additional lines of boilerplate:

- Setting `event_handled` to `true` following the context check.
- Adding a case for the new state to `stringify_state`.

Thus bringing the total boilerplate to eight lines of code per state.

As mentioned in Section 5.2.1, FSM implementations leveraging the `fsm` template have the logging logic for the three concerns detailed above already implemented as part of the template.

The coroutine designs do not provide any built-in logging support whatsoever. However, the elements of the design already provided prove almost sufficient for this purpose:

- Logging of received and unhandled events can be implemented through implementing notification methods that wrap logging logic around calls to `send_now()`, which are then used instead of `send()` and `send_now()` directly. Asynchronous sends must be done through immediate `after` statements on such notification methods.
- State transitions can be logged on an ad hoc basis directly within asynchronous logic, e.g.:

  ```
  await event_A;
  log info: "Coroutine progressed to waiting for event_B";
  await event_B;
  ```

This approach is questionable due to the involvement of non-trivial boilerplate:

- Wrapper logic around uses of `send_now()` must be implemented for each individual channel or channel array,[12] discouraging the use of multiple individual channels.
- ad hoc logging should ideally be done for each possible suspension point transition, which intrudes on the development of asynchronous logic.

The boilerplate issue of channel-specific wrapper logic can be ameliorated by minimizing the number of individual channels through channel arrays or by having channels represent multiple different events through the associated message. However, the second source of boilerplate cannot be solved with the design as is. There is, however, a minimal extension to the designs that would allow a user to gain access to a descriptor for the current suspension point of a coroutine for use in logging; namely, the means to access the serialized representation of the current suspension point, as described in Section 3.1.4. To this end, the `suspension_context()` submethod of coroutine objects is introduced, which returns a string of the serialized representation of the current suspension point, which can be used until the coroutine is next resumed.[13] The rewritten FSMs were developed assuming that this method was available.

Translating the above FSM example to use coroutines, the following demonstrates how logging can be implemented:

```
// Same as before
method stringify_event(uint64 event) -> (const char *) {
  ...
}

method log_received_event(uint64 event) {
  log info, 4, somefsm: "Received event %s at suspension context %s"
                      , stringify_event(event)
                      , somefsm_coroutine.suspension_context();
}

method log_unhandled_event(uint64 event) {
  log error, 2, somefsm: "Unhandled event %s at suspension context %s"
                       , stringify_event(event)
```

[12] DML allows users to declare arrays of objects. See Section 5.2.4 for an example of how channel arrays may be used.

[13] To elaborate, the returned string is only guaranteed to be unchanged and reference a valid memory location up until the coroutine is next resumed. This allows the string to allocated on the heap and automatically freed once the coroutine is resumed – removing that responsibility from the user, at the cost that they must use `strdup()` in order to preserve the string past coroutine resumption.

```
                      , somefsm_coroutine.suspension_context();
}

method somefsm(uint64 event) {
  log_received_event(event);

  const char* prev_context =
    strdup(somefsm_coroutine.suspension_context());
  local uint64 resumed = somefsm_coroutine.event_chan.send_now(event);
  if (resumed == 0) {
    log_unhandled_event(event);
  } else {
    log info, 4, somefsm: "Coroutine progressed: %s -> %s"
                        , prev_context
                        , somefsm_coroutine.suspension_context();
  }
  free(prev_context);
}
```

Note that no changes are needed to the body of the coroutine object: this logging is completely unintrusive for the development of asynchronous logic, and does not scale with increasing number of suspension points. This is the approach in the rewritten FSMs in order to replace generic logging of events and transitions. Any other ad hoc logging that FSM logic features is simply transcribed to the corresponding asynchronous code.

Despite the benefits of this approach, it has multiple severe issues:

- The serialized representation of coroutine suspension points – although readable – requires careful study of source code to discover what suspension point they correspond to.
- It further increases the size and complexity of the necessary wrapper boilerplate around uses of `send_now()`, causing the boilerplate to be significant even with a small number of individual channel declarations.
- The need for wrapper logic makes `send()` and `send_now()` unsuitable for direct usage in DML programs, which is a failure of the design.

As no satisfactory approach has been found for logging the progression of asynchronous logic with the existing designs, these designs must be extended with native support for such logging.

Native logging logic can be provided by coroutine and channel objects – coroutines may log about their own progression, while channels may log about received and unhandled messages. The logging behavior must be configurable, as the desired behavior may vary between use cases – in particular, what log types, log groups, and log levels are to be used. Assuming that both coroutine and channel objects are enforced to be *compound objects*, such

configurability is simple to offer through overridable logging parameters which users may provide together with the declaration of coroutine and channel objects. This has the benefit of offering users the ability to leverage the existing metaprogramming features of DML for templates and compound objects, allowing e.g. the instantiation of logging parameters for entire groups of coroutine objects or channel objects.

Configurability is significantly more difficult to support for any design which relies on non-compound coroutine and/or channel objects. Two approaches have been identified:

- Have coroutines and channels offer methods for the configuration of logging parameters. A consequence of this that logging parameters would be modified at run-time. The desirability of this is suspect – extrapolating from existing DML code studied over the course of the thesis, it is heavily anticipated that logging parameters used in a model would almost always be static. Thus, the configuration methods would only be called once, as part of initialization.

  If coroutines and channels are indeed non-compound, the code reduction techniques enabled by DML's template metaprogramming can't be applied – making the needed boilerplate to instantiate logging parameters scale with additional coroutines and channels.

- The addition of new language features in order to configure the logging of coroutines and channels statically, which would act as ad hoc replacements for the template and parameter-based solutions that compound objects allow. This would introduce additional language complexity.

Although non-compound coroutine and channel objects enable additional flexibility in design and implementation, these approaches are subject to significant drawbacks compared to what compound objects would allow. This presents a significant drawback due to the evident importance of logging.

In conclusion, compound coroutine and channel objects – even if only for logging purposes – are heavily recommended.

5.2.3 State Introspection

All states of FSMs are explicitly declared and labeled by necessity – in contrast, the coroutine designs were developed to place the suspension points of a coroutine on a higher abstraction level – they are not be labeled; their presence is simply indicated through the `await` keyword; and they are considered internal and inherently unstable. In particular, there is no native method for any component of a DML program to discover the current suspension point of a coroutine. This design decision was made in order to reduce the amount of boilerplate demanded of the user by removing the need for labeled, explicit states, and to allow the development of asynchronous

logic to be flexible – so that suspension points may easily be changed, moved, or removed, and without risking breakage in other parts of the program.[14]

The fact that states of an FSM are explicitly labeled is sometimes taken advantage of in the original modules by studying the current state of an FSM outside of that FSM's core logic. Such *state introspection* is typically used in order to determine if an FSM is in an idle or terminated state. This presents an issue for the coroutine solutions of the rewritten FSMs, as there is no native way to determine the current suspension point of a coroutine.

This issue may be solved by leveraging mutable variables in order to keep track of the status of a coroutine. The variables may be modified by the coroutine as it progresses, and studied by logic external to the coroutine. Such variables may be introduced as necessary, and can be selectively updated only for a few number of important progressions – or even just one particular property – rather than being updated for each possible suspension point. For example, a coroutine being in an idle state or not can be signaled through an `active` flag, which becomes set as the coroutine starts.

```
coroutine some_coroutine {
  channel () start;
  saved bool active = false;

  async method coroutine() {
    active = false;
    await start;
    active = true;
    ... // rest of logic
  }
}
```

Another option, when appropriate, is to make use of the `suspended()` method of a channel – in the example below, the `start` channel is exclusively used by `some_coroutine`, and so `start.suspended()` is equal to zero if and only if the coroutine is outside of its idle state.

```
coroutine some_coroutine {
  channel () start;

  method active() -> (bool) {
    return (start.suspended() == 0);
  }

  async method coroutine() {
    await start;
    ... // rest of logic
  }
}
```

[14] This does not include potential breakage to previous checkpoints.

5.2.4 Exceptional Event Handling

Exceptional events constitute events that may be accepted over a span of multiple subsequent states of an FSM, and receiving such events interrupts the regular logic of the FSM. Such events could represent reset, cancellation, or timeout signals.

Across the developed coroutine designs, this corresponds to (canceling) scoped message handling (see Section 4.4), which relies on the use of subcoroutines.

This pattern is rare – only three of the 28 FSMs feature it. In all instances, such messages represent a form of reset or cancellation signal that the FSM must accept in any state, and enter a specialized state chain in order to handle the post-reset procedure. These instances can be converted to coroutine code by declaring a subcoroutine for the main logic flow, together with a small number of subcoroutines for reset/cancellation logic flow – the main coroutine logic simply consists of waiting for the start/reset signals, and starting, canceling, or restarting the appropriate subcoroutines.

Expanding upon the `switch`-based pattern presented in Section 5.2.1, exceptional event handling typically manifests as follows:

```
param EVENT_A = 0;
param EVENT_B = 1;
param EVENT_RESET_TYPE_A = 2;
param EVENT_RESET_TYPE_B = 3;
...

param STATE_A = 0;
param STATE_B = 1;
...

saved uint64 curr_state = STATE_A;

// Notification method, containing the core FSM logic.
// Notification must be coupled with info about event kind.
method somefsm(uint64 event) {
  switch(event) {
  case EVENT_RESET_TYPE_A:
    on_type_a_reset();
    curr_state = STATE_RESET_A_1;
    return;
  case EVENT_RESET_TYPE_B:
    on_type_b_reset();
    curr_state = STATE_RESET_B_1;
    return;
  default:
    break;
  }
```

```
  switch(curr_state) {
  case STATE_A:
    if (event == EVENT_A) { // Context check
      ... // react to EVENT_A
      curr_state = STATE_B;
    }
    break;
  ...
  case STATE_RESET_A_1:
  ...
  case STATE_RESET_B_2:
  }
}
```

This usage pattern has revealed unanticipated difficulties in the various designs – specifically, the issue lies in that, with these semantics, a specific exceptional event may not only simply interrupt the main logic of the FSM – it may interrupt the FSM flow for handling a previously received exceptional event – *potentially of the same event kind*, causing it to restart. These semantics make it difficult to use `async` method calls as triggers (or the bounded design's corresponding `run()` submethod of subcoroutines) in order to perform scoped message handling – as the handlers themselves need to have exactly the same message handlers attached to them. This is not possible to express in any developed design without difficulty.

Unexpectedly, this issue is easier to resolve for the bounded design rather than the unbounded design, by virtue of all subcoroutines always being statically available. The above code can be translated as follows:[15]

```
param EVENT_A = 0;
param EVENT_B = 1;
param EVENT_RESET_TYPE_A = 2;
param EVENT_RESET_TYPE_B = 3;
...
param TOTAL_EVENTS = ...;

coroutine somefsm_coroutine {
  channel () events[TOTAL_EVENTS];

  subcoroutine main() {
    // Corresponds to STATE_A
    await events[EVENT_A];
    ... // react to EVENT_A
    ... // Rest of converted FSM logic
  }

  subcoroutine reset_a() {
```

[15]The usage of channel arrays differs from the conditional listen-based solution presented in Section 5.2.1

```
    on_type_a_reset();
    ... // FSM logic for reset of type A.
  }

  subcoroutine reset_b() {
    on_type_b_reset();
    ...// FSM logic for reset of type B.
  }

  method cancel_subcoroutines() {
    main.cancel();
    reset_a.cancel();
    reset_b.cancel();
  }

  async method coroutine() {
    cancel_subcoroutines();
    main.start();

    while (true) race {
      case (await events[EVENT_RESET_TYPE_A]) {
        cancel_subcoroutines();
        reset_a.start();
      }
      case (await events[EVENT_RESET_TYPE_B]) {
        cancel_subcoroutines();
        reset_b.start();
      }

      race {
        case (await main);
        case (await reset_a);
        case (await reset_b);

        // Can be omitted if each of the possible
        // logic flows have a terminal state.
        completed {
          break;
        }
      }
    }
  }
}

method somefsm(uint64 event) {
  somefsm_coroutine.events[event].send_now();
}
```

This approach is notably verbose – indeed, more so than the FSM solution. Although it is possible to reduce the code size somewhat through different

approaches – for example, unifying the bodies of `reset_A` and `reset_B`, and pass a parameter to chose the reset type – the above represents what most closely corresponds to the intended uses of subcoroutines and approach to scoped message handling.

The above approach *can't* be applied to the unbounded design, due to following aspects:

- Subcoroutines are dynamically created; they can't be referenced statically.
- The lifespans of dynamically created subcoroutines are limited to a local scope.
- Dynamically created subcoroutines can't be restarted.

Because of this, the unbounded design must either recreate static subcoroutines using the same pattern as in the basic design, described in Section 4.3, or use an alternative means to control which of the possible logic flows are executed. The following leverages participant guards together with a mutable variable to this end:

```
param EVENT_A = 0;
param EVENT_B = 1;
param EVENT_RESET_TYPE_A = 2;
param EVENT_RESET_TYPE_B = 3;
...
param TOTAL_EVENTS = ...;

coroutine somefsm_coroutine {
  channel () events[TOTAL_EVENTS];

  param PHASE_MAIN    = 0;
  param PHASE_RESET_A = 1;
  param PHASE_RESET_B = 2;

  async method main() {
    // Corresponds to STATE_A
    await events[EVENT_A];
    ... // react to EVENT_A
    ... // Rest of converted FSM logic
  }

  async method reset_a() {
    on_type_a_reset();
    ... // FSM logic for reset of type A.
  }

  async method reset_b() {
    on_type_b_reset();
```

```
    ...// FSM logic for reset of type B.
  }

  async method coroutine() {
    saved local uint8 phase = PHASE_MAIN;
    while (true) race {
      case (await events[EVENT_RESET_TYPE_A]) {
        phase = PHASE_RESET_A;
      }
      case (await events[EVENT_RESET_TYPE_B]) {
        phase = PHASE_RESET_B;
      }

      race {
        if (phase == PHASE_MAIN)    case (await main());
        if (phase == PHASE_RESET_A) case (await reset_a());
        if (phase == PHASE_RESET_B) case (await reset_b());

        // Can be omitted if each of the possible
        // logic flows have terminal state.
        completed {
          break;
        }
      }
    }
  }
}

method somefsm(uint64 event) {
  somefsm_coroutine.events[event].send_now();
}
```

This pattern can easily be converted for use with the bounded design, and its verbosity is comparable to the solution relying on statically declared subcoroutines.

The severity of these issues depends on how often the expressed semantics are needed. Although the rewritten FSMs do indeed possess these semantics, it's possible that these are only an artifact of the simplicity of the pattern expressed in the original FSM logic; if, for example, the relevant reset signals aren't repeated when already being handled, then there is no need to replicate the precise semantics of the original FSM, allowing for simpler implementations.

Nonetheless, the difficulty of expressing the usage pattern does expose flaws in the developed designs – in particular, the unbounded design. The dynamic subcoroutines of the unbounded design were intended to subsume all uses of the static subcoroutines of the bounded design; but the above issues with the unbounded design show that this is not the case.

In conclusion, the forms of scoped message handling that the designs were developed to address do not seem to be represented within the studied asynchronous logic. The scoped message handling that is present exposes weaknesses in the designs – in particular, the unbounded design. This is not a definite failure of the designs, but does show that the designs for subcoroutines are flawed, and must be reevaluated and reworked.

5.2.5 External State Transitions

An external state transition is when logic external to the core logic of an FSM forces a transition to a particular state, no matter the current state of the FSM. This is similar to exceptional events, and is used for the same purposes (e.g. resets) – the difference lies in that external state transitions change the state of the FSM directly, rather than informing the FSM of the event. The ability to perform external state transitions is an artifact from the need of FSMs to have explicitly labeled states and globally accessible storage location for the current state of an FSM – just as state introspection (see Section 5.2.3).

With the exception of moving execution of a coroutine to its beginning (through the `restart()` method), such external state transitions are not natively supported by any developed coroutine design, nor is there an intuitive solution as with state introspection.

Two approaches have been developed in order to address this usage pattern:

(a) It may be possible to represent external state transitions as (exceptional) events, instead – which are messaged to the coroutine, which acts accordingly in order to move execution to the proper suspension point. This is the most idiomatic solution, when applicable, as it removes the need to expose labeled states of the coroutine – instead abstracting it as a command.

This approach is most appropriate if the external state transitions can only happen within specific ranges of the coroutine's execution, and such transitions only move execution to a future point of the coroutine's logic, or to a wholly different line of logic that would otherwise never be executed. However, it is an untenable solution if the coroutine must be able to move execution to a *prior* point of its logic (except for its beginning, which is achievable through the `restart()` method), or if the forced transition can occur both *before* and *after* the target point.

(b) External state transitions can be communicated to a coroutine by forcing its execution to a centralized point, from which it determines and moves execution to the desired target suspension point.

Forcing execution to a central point can be accomplished either through a scoped message handler over the coroutine as a whole or through

`restart()`. The execution move can be accomplished through `switch`es and/or `if-else` statements. Switch statements in particular can be useful in order to label the necessary target suspension points, as fall-through can be leveraged in order to reduce the verbosity of moving execution to the correct point without otherwise disturbing the normal structure of asynchronous code.

The following showcases how the second approach can be accomplished:

```
param PHASE_IDLE = 0;
param PHASE_OPERATE = 1;
param PHASE_RESET = 2;

coroutine cor {
  session uint8 start_point = PHASE_IDLE;

  method force_to_point(uint8 target_point) {
    start_point = target_point;
    restart();
  }

  async method coroutine() {
    local uint8 startat = start_point;
    start_point = PHASE_IDLE;
    switch (startat) {
    case PHASE_IDLE:
      await start;
      // FALLTHROUGH
    case PHASE_OPERATE:
      while (true) {
        local uint8 done = await do_operate();
        if (done) break;
      }
      break;
    case PHASE_RESET:
      await do_reset();
      break;
    }
  }
}
```

The ability to perform external state transitions is offered through the `force_to_point()` method, which restarts the coroutine – bringing it to the centralized point at which it identifies and transitions to the target state – and uses a mutable variable in order to communicate the target state. This requires the reintroduction of labeled suspension points, but unlike FSMs, not every unique suspension point needs to be labeled – only the ones which external state transitions need to target. Hence, these are called *phases*, rather than *states*, in the example code above.

The need for this pattern is very rare. Two of the 28 FSMs had an external state transition to one particular state beyond the initial state – this could implemented through a simple boolean variable and an `if` statement rather than constants and `switch` statements. Only one FSM had external state transitions to more than one state beyond the initial state, for which the above pattern was applied.

As shown above, the code impact of this solution is moderate. The most significant issue of this approach is that the pattern itself is not intuitive, which is deemed acceptable due to how rarely it is needed.

5.2.6 Structured Non-Linear Asynchronous Logic

The developed coroutine designs do not offer the same flexibility as traditional FSMs in transitioning between suspensions points. Asynchronous logic under these designs relies on structured programming in order to specify control flow, which corresponds to linear logic that allows for limited non-linear control flow through the use of conditional statements, loop statements, and `race`/`concurrently` statements. These features are appropriate for one of the following use cases:

- A line of asynchronous logic which may conditionally be *skipped* over to a later suspension point. (`if` statements)
- A line of asynchronous logic to be repeated until a condition is met. (loop statements)
- Multiple possible lines of asynchronous logic that may be entered at a certain point, but have unified exit points. (`if-else`, `switch`, and `race`/`concurrently` statements).

Non-linear asynchronous logic is common:

- 39 states out of the total 277 contain logic for conditionally *skipping* to a later point of the FSM logic.
- Three states corresponded to loops which were exited when a condition was met.
- In addition to the 28 states accepting multiple different events as detailed in Section 5.2.7, One state contained different lines of asynchronous logic being executed depending on a condition independent of the received event. (`if-else`)

The extent to which these patterns reduce boilerplate varies, but largely mirrors the reduction observed for simple linear states.

5.2.7 Waiting for Multiple Events

The typical mode of operation for each studied FSM is to only accept one particular event in a given state – however, a common pattern is the need to be able to handle multiple different events in a state, or to wait for multiple events to occur in a state before progressing. Such states correspond to usages of `race` and `concurrently` under the developed coroutine designs, and the occurrence rate of such states and improvement in code quality when rewritten to use `race`/`concurrently` is indicative of the success of these features.

Across the subset of 18 FSMs with explicit event kinds, 28 states out of 229 total accepted multiple different event kinds – 12%; and of those, 13 states corresponded to concurrently waiting for a set of events to transpire before resuming execution.

All but 3 of these 28 states were converted to make use of `race`/`concurrently`. The extent to which this improves code quality varies:

- FSM code typically leverages `switch` statements in order to handle different events. The resulting pattern is similar to the use of `race`, making overall code quality is similar. This is considered satisfactory, as the original FSM pattern is concise and easily understandable.

  The FSM approach is as follows:

```
case STATE_A:
  switch (event) {
    EVENT_A:
      ... // React to EVENT_A
      break;
    EVENT_B:
      ... // React to EVENT_B
      break;
    EVENT_C:
      ... // React to EVENT_C
      break;
  }
  break;
```

  The coroutine approach using `race` is as follows:[16]

```
race {
  case (await events[EVENT_A]) {
    ... // React to EVENT_A
  }
  case (await events[EVENT_B]) {
    ... // React to EVENT_B
  }
```

[16] Utilizes channel arrays

```
  case (await events[EVENT_C]) {
    ... // React to EVENT_C
  }
}
```

- The FSM logic to represent a state concurrently waiting for a set of events to complete involves significant amounts of boilerplate. The conventional approach is to use a set of mutable variables to keep track of event completion, as follows:

```
saved bool event_A_received = false;
saved bool event_B_received = false;
saved bool event_C_received = false;

...

method somefsm(uint64 event) {
  switch(curr_state) {
  ...
  case STATE_A:
    switch (event) {
      EVENT_A:
        event_A_received = true;
        break;
      EVENT_B:
        event_B_received = true;
        break;
      EVENT_C:
        event_C_received = true;
        break;
    }
    if (   event_A_received
        && event_B_received
        && event_C_received) {
      // waiting done
      event_A_received = false;
      event_B_received = false;
      event_C_received = false;
      ... // react to EVENT_A, EVENT_B, and EVENT_C being completed
    }
    break;
  ...
  }
}
```

  Note the need to:

  - Prepare mutable variables representing event reception
  - Specify handlers for each event and set the corresponding variable

– Perform a check for if each event has been received

In contrast, the corresponding coroutine code leveraging `concurrently` is as follows:

```
coroutine somefsm_cor {
  async method coroutine() {
    ...
    concurrently {
      case (await events[EVENT_A]);
      case (await events[EVENT_B]);
      case (await events[EVENT_C]);
    }
    ... // react to EVENT_A, EVENT_B, and EVENT_C being completed
  }
}
```

The `race` and `concurrently` statements are deemed to be satisfactory of the design. These statements fulfill a unique – but commonly expressed – need in asynchronous logic, and may be leveraged in a concise manner. `race` usage is similar to the quality of `switch` patterns, while `concurrently` allows for greatly reducing boilerplate.

Participant guards are leveraged very rarely in the rewritten modules (three times) but have proven valuable in order to implement more complex asynchronous logic. No matter their rarity, their omission is not recommended, due to the extreme difficulties in implementing alternate solutions once conditionally included participants are needed. In the worst case, it would be necessary to have separate `race`/concurrently statements for each possible combination of participants that may be part of a `race`/`concurrently`, and then use `if-else` statements to chose the correct `race`/`concurrently`; leading to an exponential growth in code size as the number of optional participants grows, which is unacceptable.

`completed` declarations were occasionally leveraged in order to execute common code following a `race` – but were never used for their intended primary usage within `concurrently` participants. This is because such participants were never necessary – every use of `concurrently` in the rewritten modules could not be interrupted by other events.

5.2.8 Reset Procedures

Reset procedures are procedures that describe how the state of FSMs should be reset, and are typically invoked upon the parent device receiving a particular reset signal. Such reset procedures are typically based upon DML's built-in reset templates (hard reset, soft reset, power-on reset). Coroutine objects under the basic design automatically provide support for these forms of reset, restarting the coroutine upon any kind of reset. However, the

benefits of this support are unclear, and making reset handling *opt-in* rather than opt-out has the benefit of enabling non-compound coroutine objects, as described in Section 4.2.2. It was thus of interest to study the use of built-in reset procedures during evaluation.

All FSMs studied either made use of *registers* or *attributes* in order to store the current FSM state.[17] Registers have built-in reset handling – which must be explicitly disabled or overwritten – whereas attributes do not. Note that this does *not* concern FSMs with reset events as part of their logic (typically represented through exceptional events).

Out of the 28 FSMs studied:

- 24 did not make use of built-in reset procedures, or their parent device did not implement any reset signals.
  - If registers were used to store FSM state, then these were explicitly overridden to ignore resets.
  - If attributes were used to store FSM state, then no means were provided to reset their state as part of any reset procedure of the parent device.
- Four made use of simple or default built-in reset procedures that simply reset the FSM to its initial state.
  - If registers were used to store FSM state, then their reset behavior was not overridden – except potentially to specify the initial value.
  - If attributes were used to store FSM state, then reset procedures to reset the FSM to its initial state were provided for all reset kinds in use.
- Zero involve non-trivial reset logic, potentially involving different actions for different event kinds, or performing logic that does not always reset the coroutine.

This reveals that the vast majority of the FSMs under study do not make use of built-in reset procedures, and so the benefit of having built-in reset logic is minimal. This negates the drawback of non-compound coroutine objects identified in Section 4.2.2.

5.2.9 Cross-State Variables

Cross-state variables represent (mutable) data that needs to persist between activations of an FSM over a range of states, and aren't used outside of the asynchronous logic of that FSM. Such variables may be implemented through

[17] At the time of the thesis project `saved` variables were a relatively new feature, and the studied codebases did not make any significant use of it.

`session` or `saved` variables, or by `register` or `attribute` objects. Cross-state variables correspond to preserved local variables under the developed coroutine designs; `saved local` or `saved` variables.

Cross-state variables are rare in the original FSM modules; it is significantly more common for mutable data to be shared with other components of the device. Only two FSM featured cross-state variables, resulting in a total of five cross-state variables.

Usage of `saved` and `saved local` in the rewritten modules typically don't correspond to cross-state variables; instead, these are usually being introduced in order to implement complex asynchronous logic. The greatest value of such variables is for `async` methods to preserve received parameters past suspension points. This has been leveraged in the rewritten modules in order to separate verbose uses of conditional listens to methods, as detailed in Section 5.3.

In conclusion, preserved local variables rarely correspond to patterns in FSM logic, due to the rarity of cross-state variables in the original FSMs. Their value is thus mostly dependent on their usefulness in novel coroutine code; particularly as a means for `async` methods to preserve received parameters. If the need for such `async` methods proves to be rare, then `saved local` variables can be omitted from modifications upon the unbounded design.

5.2.10 Delays and Timeouts

The `await delay` statement was developed in order to address the need for FSMs to establish timeouts for certain events or in order to delay its own progression.

Across the 28 rewritten FSMs, only three FSMs featured use cases that could be converted to `await delay` expressions with non-zero delay; two further rewritten FSMs made use of immediate `delays` in order to control execution order, making `await delay` one of the more underutilized features of the basic design.

Although FSMs commonly feature artificial latency, it is typically inserted at *event transmission*; events are delayed before being sent to the FSM. `await delay` can't be applied for this purpose, since event transmission takes place outside of asynchronous logic.

The rarity of `await delay` being applicable was unexpected, making the value of its inclusion unclear; however, due to its low cost of inclusion, simplicity, precedence in other languages, and obvious and seemingly desirable use cases, there is very little reason for it to be discarded.

5.2.11 Terminal States

A terminal state of an FSM is a state from which the FSM will not progress as part of its regular logic. In order for the FSM to be able to continue

operation, either an exceptional event or an external state transition must take place – typically, a reset bringing the FSM to its initial state.

Under the iterated coroutine designs, a terminal state corresponds to a suspension point for which the local asynchronous logic only permits progression to the same suspension point, or doesn't permit progression at all. This can be implemented through a loop over the same suspension point which is never exited, or by suspending upon a channel which intentionally is never sent messages. Such suspension points can only be exited through a scoped message handler being triggered or by restarting the coroutine – which corresponds to exceptional events or external state transitions.

Coroutine objects under the basic design have an implicit terminal state at the end of their associated `coroutine()` method; but the iterated coroutine designs do not, instead looping execution of the `coroutine()` method forever. This is because the judgment was made that the latter behavior would reflect almost all real-world use cases of coroutines, as discussed in Section 4.2.1; however, this was not observed during evaluation. Out of the 28 FSMs rewritten, 17 have a terminal state as part of their normal execution flow in which they are not able to handle any events – 61% of all FSMs. This warrants a reexamination of what the default behavior of coroutines should be. Although the majority of FSMs possess terminal states, it may be argued that the code impact of explicitly looping the execution of a coroutine compared to explicitly introducing a terminal state is significantly greater – thus justifying implicit looping of coroutines.

- If coroutines have an implicit terminal state, then looping the logic of a coroutine may be done through a `while (true)` scoping over the entire `coroutine()` method – which introduces another indentation level for asynchronous logic:

  ```
  coroutine explicitly_looping_coroutine {
    async method coroutine() {
      while (true) {
        ... // Asynchronous logic
      }
    }
  }
  ```

  Alternatively, the indentation level can be decreased by moving the main coroutine logic to a separate `async` method, which is called by `coroutine()`:

  ```
  coroutine explicitly_looping_coroutine {
    async method coroutine() {
      while (true) {
        await logic();
      }
    }
  ```

```
  async method logic() {
    ... // Asynchronous logic
  }
}
```

- If coroutines implicitly loop forever, introducing a terminal state can be done through declaring a `channel` to which messages are never sent, and then suspending upon that channel as the last statement of the asynchronous logic.

```
coroutine explicitly_terminal_coroutine {
  channel () never;

  async method coroutine() {
    ... // Asynchronous logic
    await never;
  }
}
```

Because of this, there is no definite conclusion as to which approach is of greater value; either is justifiable, and so no specific recommendation is made.

5.3 Miscellaneous Details

There are several notable details regarding the use of each design unrelated to their application for the key usage patterns identified in the original FSMs.

5.3.1 Use of `async` Methods

The restrictions placed on `async` methods and their use in the bounded designs compared complicate their usage, and as a consequence the bounded design occasionally needs to seek alternate approaches which are not necessary in the unbounded design. As a particularly notable example, consider the use of conditional listens in order to translate FSMs with explicit event kinds; waiting for a particular `TARGET_EVENT` is then implemented as follows:

```
local uint64 event = await event_chan
                     where (event == TARGET_EVENT);
```

This pattern is verbose, making it appealing to separate this code to a separate `async` method. In the unbounded design, this method can be idiomatically written as follows:

```
async method wait_for_event(saved uint64 desired_event) {
  local uint64 event = await event_chan
                       where (event == desired_event);
}
```

However, the bounded design does not feature saved local variables. In order to preserve the constant-valued parameter, the following is required:

```
async method wait_for_event(uint64 _desired_event) {
  saved uint64 desired_event;
  desired_event = _desired_event;
  local uint64 event = await event_chan
                       where (event == desired_event);
}
```

Which is unintuitive, and prevents the method from safely being used by multiple coroutines – in particular, between multiple subcoroutines – due to the shared `saved` variable. In addition, the bounded design does not support `async` method calls as triggers within `race`/`concurrently`, thus requiring the verbose syntax if the coroutine must be able to accept multiple event kinds at a suspension point. In order to resolve this issue, rewritten modules under the bounded design typically leverage *channel arrays* instead, as shown in Section 5.2.4. Although this approach can typically be applied without issue, it relies on that the range of constants used to represent event kinds are suitable for use as indices – that is, the constants are contiguous integers starting from zero. This is true for most of the rewritten FSMs, but some featured large gaps between the constants used.

Although neither saved local variables nor `async` method calls as triggers were used to their full, intended potential, their support in the unbounded design enabled the idiomatic use of `async` methods, allowing for compartmentalization and reuse of arbitrary asynchronous code without issue.

5.3.2 Negligible Impact of Non-compound Coroutine Objects

Despite the fact that non-compound coroutine objects allow for decreased boilerplate, this rarely manifested in practice. Coroutine code is often grouped together with other logic and resources related to the coroutine as part of a compound object. This was occasionally done even when the original asynchronous logic manifested on the top-level in the original FSM code, as it allows for defining methods and channel objects which may be easily referenced within the coroutine code without possibly disrupting the global name-space. E.g. with channel arrays:

```
param EVENT_A = 0
...
param TOTAL_EVENTS = ...


coroutine somecoroutine {
  channel () events[TOTAL_EVENTS];

  async method coroutine() {
```

```
    await events[EVENT_A];
    await events[EVENT_B];
    ...
  }
}
```

`events` is a convenient identifier for the channel array, but the generic nature of the name makes name collisions plausible – necessitating it being restricted within the scope of a compound object.

With non-compound coroutine objects, the above manifests as:

```
param EVENT_A = 0
...
param TOTAL_EVENTS = ...


bank somecoroutine {
  channel () events[TOTAL_EVENTS];

  coroutine logic {
    await events[EVENT_A];
    await events[EVENT_B];
    ...
  }
}
```

in which case non-compound coroutines do not provide any benefit.

5.4 Discussion

5.4.1 Reliability and Potential Biases

Despite the reliability issues of the code reduction metrics as noted in Section 5.1, it is plausible that the result of ~ 7.70 lines reduced per state for the restricted subset of 26 modules do reflect the performance of coroutine designs on a larger scale. The reasoning in Section 5.2.1 and Section 5.2.2 shows that a reduction of 7 lines of code per state can be expected for the simplest – and most common – forms of asynchronous logic. This figure is further increased by the comparatively concise syntax of `concurrently` and structured asynchronous logic. The usage patterns for which the coroutine designs have performed poorly – such as exceptional events and external state transitions – are rare, and thus do not significantly affect the figure. Nevertheless, it is unclear if the results can be extrapolated due to multiple potential biases in the evaluation process.

The most notable bias lies in the selection of FSM modules – easily understood modules with clear, compartmentalized FSM logic were prioritized, while modules featuring unclear FSM control flow were typically discarded. This choice was made in order to have as high throughput of rewritten FSMs

as possible, but may cause the sample set to be unrepresentative. The FSMs under study often shared code patterns, source package, author, and/or pieces of logic; they can't be considered independently chosen samples. This bias can be eliminated by either an exhaustive rewrite of all FSMs present in the utilized codebases, or through random selection – either of which were not valid options during this thesis due to time constraints involved.

Another potential bias lies in differences between expressive power of traditional FSM approaches and the developed coroutine designs – although coroutines under the various designs are not as flexible as manually implemented FSMs, they are able to offer abstractions which are impossible succinctly express under FSMs. Because of this, direct transcription of FSM code typically maps only to the core elements of the coroutine designs, leaving other elements possibly underrepresented.

For example, `async` methods are effectively an abstraction for injecting new suspension states into a coroutine – which corresponds to injecting sets of states into an FSM. This allows for improved code compartmentalization and reuse, which is occasionally leveraged in the rewritten modules – but it also enables the development of interfaces that offer asynchronous methods as part of its API. As `async` methods have no idiomatic corresponding solution in traditional FSMs, the usage patterns `async` methods would enable are never expressed in the source modules, even though there is precedence for such use cases in other languages. SystemC infrastructure heavily relies on interfaces offering asynchronous functions – in particular, interfaces offering `b_transport()`.

This bias can only be eliminated by developing novel coroutine code, rather than rewriting existing FSM code into coroutines. This would provide much greater insight into what elements of the developed designs prove valuable for the development of asynchronous logic.

5.4.2 Conclusions Regarding Developed Designs

With the only exception of `await delay` expressions, the core elements of the coroutine designs – as described by the basic design – have performed extremely well. These elements are ubiquitously used within the rewritten modules,[18] and dramatically reduce boilerplate when applied.

In contrast, the elements introduced by the various iterated designs proved rarely applicable – and occasionally suffered from usability issues when they were. Subcoroutines were only leveraged across three FSMs – with siz subcoroutine declarations total – and the design for dynamic subcoroutines within the unbounded design has shown significant issues.

[18] This is only significant for elements that reflect specific needs and use cases – such as `race` and `concurrently` statements. The prevalence of such elements reflects how common the corresponding use cases are. In contrast, elements such as `coroutine` and `channel` are mandatory in order to leverage any design; they are always present by necessity.

Although subcoroutines are still believed to have merit due to their flexibility and ability to resolve issues identified during iteration:

- Their need is rare enough to possibly justify dropping native support, instead requiring users to implement them ad hoc as per the approach demonstrated in Section 4.4.2.
- The native support offered by the iterated designs has significant usability issues, requiring redesign.

Because of this, the recommended course of action is to *not* adopt explicit native support for subcoroutines – instead leaving them as a potential extension to the design, to be included or revised once the significant usability gaps of the coroutine designs have become better understood.

The most significant usability gap identified across all coroutine designs is the lack of native support for logging of messages sent to channels as well as coroutine progression, as detailed in Section 5.2.2. This must be addressed, as the amount of boilerplate that would otherwise be necessary would heavily discourage increased number of channels, as well as the use of shared channels. The recommended approach is to allow the logging of `coroutine` and `channel` objects to be configured via `params` – which requires that `coroutine` and `channel` are both compound object types.

The design forks featuring non-compound coroutine objects should be abandoned; even though opt-in reset handling has been determined to be unproblematic, non-compound coroutine objects provide little benefit, and severely complicate potential solutions to the logging issue.

6 Conclusion

A detailed specification for the basic coroutine design has been presented, together with details of how this design is iterated upon to form the *bounded* and *unbounded* coroutine designs. The developed designs are primarily centered around statically declared *coroutine* and *channel objects* – coroutine objects represent instances of coroutines, which may suspend their execution by listening to channels. A coroutine suspended on a channel is resumed when another part of the device model *sends a message* to that channel – which causes that message to be propagated to the resumed coroutine.

These designs have been evaluated by selecting 28 existing implementations of asynchronous logic in DML leveraging finite state machines, which were then rewritten to leverage coroutines; the success of each design can be gauged by their impact on the code. The selection of FSMs was heavily biased towards modules with easily understood FSM structure and control flow, in order to allow for a greater number of FSMs to be rewritten. As a result, the results are indicative of how well the coroutine designs perform when used to implement asynchronous logic of low to medium complexity,

but it is unclear whether the results can be extrapolated to asynchronous logic in general.

The rewritten modules show significant improvements in code quality for each of the developed coroutine designs compared to the existing state machine implementations. Code size is reduced by an average of:

- 9.6 lines of code per unique state across the original state machines;
- 7.7 lines of code per unique state, when excluding a small number of modules that are dramatically affected by abnormal usage patterns;
- 7.2 lines of code per unique state, when only studying the simplest forms of asynchronous logic.

Although most elements of the basic design were heavily utilized, the extensions featured in each of the iterated designs proved rarely applicable. This results in usage of each developed design to be very similar – centered around the core elements of the basic design. In addition, some of the elements of the iterated designs demonstrated unforeseen usability issues even when they were applicable.

These results potentially indicate that the additional elements provided by the iterated design are not of significant practical value; however, it is also possible that the rewritten modules underrepresent the usability of the more complex elements. The relatively low-level nature of the original state machine implementations creates a natural bias towards the primitive elements of the coroutine designs. This bias can be eliminated by evaluating novel coroutine code, developed with access to the full power of the coroutine designs, rather than rewriting existing FSM code.

Evaluation also revealed a small number of concerns which were not addressed during the development of the coroutine designs. Although most of these concerns were able to be satisfactorily resolved by leveraging existing elements of DML and the developed designs, the most significant unaddressed concern – the need to *log messages* regarding the progression of asynchronous logic – was not. This need was observed in nearly every module studied, and although such logging is possible to implement with the developed coroutine designs, it incurs significant amounts of boilerplate – which is unacceptable considering how common the need for such logging is. This issue may be addressed by providing native support for such logging while allowing users to configure how such logging is done.

In conclusion, the thesis project has succeeded in developing a core design for coroutine support in DML, which demonstrates an excellent ability to improve code conciseness when used to rewrite real-world code. However, the various extensions upon this design that have been developed require further refinement and study, in order to better understand the practical worth of these extensions, and in order to identify and address further usability issues in the developed designs.

References

[1] ISO/IEC 2382: 2015. *Information technology - Vocabulary*. 2015.

[2] ISO/IEC TS 22277:2017. *Technical Specification — C++ Extensions for Coroutines*. 2017.

[3] Accellera Systems Initiative. *About SystemC*. [Accessed 20-June-2021]. 2021. URL: https://accellera.org/community/systemc/about-systemc.

[4] *API Reference Manual – Simics® Documentation*. Version 6, Revision 6097. Intel. Apr. 2021.

[5] Bruce Belson, Jason Holdsworth, Wei Xiang, and Bronson Philippa. "A survey of asynchronous programming using coroutines in the Internet of Things and embedded systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.3 (2019), pp. 1–21.

[6] *DML 1.4 Reference Manual – Simics® Documentation*. Version 6, Revision 6097. Intel. Apr. 2021.

[7] "IEEE Standard for Standard SystemC Language Reference Manual". In: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)* (2012). DOI: 10.1109/IEEESTD.2012.6134619.

[8] Ana Lúcia De Moura and Roberto Ierusalimschy. "Revisiting coroutines". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31.2 (2009), pp. 1–31.

[9] Yury Selivanov. *PEP 492 – Coroutines with async and await syntax*. [Accessed 27-June-2021]. 2015. URL: https://www.python.org/dev/peps/pep-0492/.

[10] *Simics® User's Guide – Simics® Documentation*. Version 6, Revision 6097. Intel. Apr. 2021.

[11] *SystemC Checkpoint Library – Simics® Documentation*. Version 6, Revision 6097. Intel. Apr. 2021.

[12] S Tucker Taft, Robert A Duff, Randall L Brukardt, Pascal Leroy, and Erhard Ploedereder. *Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995 (E) with Technical Corrigendum 1 and Amendment 1*. Vol. 4348. Springer Science & Business Media, 2006.

Appendices

A Coroutine Approach to I2C Communication

```
coroutine main_thread {
  channel ()  start;
  channel ()  acknowledge;
  channel uint8 read_response;

  async method coroutine() {
    while (true) {
      await start;
      control.busy.set(1);
      i2c_link.start(Reader_Address);
      await acknowledge;
      initialize_writing();

      while (!is_writing_done()) {
        local uint8 packet = make_packet();
        i2c_link.write(packet);
        await acknowledge;
      }

      i2c_link.stop();
      i2c_link.start(Writer_Address);
      await acknowledge;
      while (!is_reading_done()) {
        i2c_link.read();
        local uint8 packet = await read_response;
        process_read(packet);
      }
      control.busy.set(0);
    }
  }
}

port i2c_in {
  implement i2c_master_v2 {
    method acknowledge(i2c_ack_t ack_value) {
      if (ack_value == I2C_ACK) {
        after: main_thread.acknowledge.send();
      } else {
        log error: "no_ack received";
      }
    }

    method read_response(uint8 response) {
      after: main_thread.read_response.send(response);
```

```
    }
  }
}

method activate() {
  main_thread.start.send();
}
```

B Detailed Semantics for `race` and `concurrently` Statements

Execution of a `race`/`concurrently` statement is as follows:

(a) Evaluate the guards of every participant, in declaration order – top to bottom, depth-first recursively in the case of nested `race`/`concurrently` participants. Every invalid participant is removed from the race. If no valid participants remain, execution of the statement completes.

(b) Traverse all remaining valid participants in declaration order, top-to-bottom, depth-first recursively, and for each triggered participant:

- If the trigger is an unconditional or conditional listen, the executing coroutine is added to the queue of the referenced channel.
- If the trigger is an `await delay` expression, then the executing coroutine is added to the queue of its anonymous `await delay` channel, if not already in it. An event is then posted to send a message identifying the current participant as triggered to the anonymous channel after the time specified by the `await delay` expression. If the `await delay` expression lacks parameters (is an immediate `delay`), then the immediate `after` queue is used to delay the send instead of posting an event.

(c) Once the coroutine is resumed, identify which participant is waiting on the specific channel that caused execution to be resumed, and attempt to resolve that participant's trigger.

(i) If the sent message is rejected, then the coroutine is suspended, and step c is repeated. The triggered participant is considered completed, and a check is performed for which of the `race`/`concurrently` statements/participants should subsequently be considered completed (if any), and what other remaining participants would subsequently leave the `race`/`concurrently` statement as a result (if any). The invalidated participants are *canceled*: the coroutine removes itself from the queue of every channel awaited upon by the canceled triggers.

(ii) Whatever the result of the check, it is followed by the execution of the body of the triggered participant. If the execution of the body would lead execution to leave the `race`/`concurrently` statement abnormally – e.g. due to a `return` statement or unhandled exception – then all remaining participants are *canceled* as above.

(iii) Once the body of the triggered participant completes normally, if the `race`/`concurrently` statement is considered completed, then execution of the statement ends. Otherwise, the coroutine is suspended and step c is repeated.

Whether or not a `race`/`concurrently` statement/participant should be considered completed is dependent on if its participants have completed:

- A `race` statement/participant is considered to be completed if any of its immediate participants are considered to be completed, or if there are no valid immediate participants. When a `race` participant is becomes canceled or is considered completed, any remaining participants within it are *canceled*.
- A `concurrently` statement/participant is considered to be completed if all of its immediate participants are considered to be completed, or if it contains no valid immediate participants. If a `concurrently` becomes canceled, any remaining participants within it are also canceled.
- An `await` participant is considered to be completed once the trigger resolves – once a message sent to the referenced channel is accepted by the executing coroutine.

C Anomalous usage patterns in FSM Modules

C.1 Event Propagation

One FSM module making use of the FSM template abstraction featured code to propagate events received by that FSM to subordinate FSMs in other modules. This was done by handling the relevant events at the few states where they are expected, and sending these to the relevant FSM(s); for example:

```
group state_a is fsm_state {
  group event_a is fsm_event_handler {
    otherfsm.events.event_a.run_now();
  }

  group event_b is fsm_event_handler {
    otherfsm.events.event_b.run_now();
    differentfsm.events.event_b.run_now();
```

```
  }

  ...
}
```

Due to the sheer number of events propagated this way, the boilerplate incurred through this alone amounted to 224 lines of code in an FSM containing 18 states – corresponding to 12.4 lines per state.

This boilerplate could be eliminated entirely when the module was rewritten under the various coroutine designs, by having the parent coroutine and its subordinate coroutines share the channels used to signal the relevant events – making propagation unnecessary.

C.2 Redundant State Transition Logic

One FSM module separated its asynchronous logic and notification method into two components:

- `update_state_handler()` – the notification method. This performs context checks dependent on the current state and – if these are successful – calculates a new *target state* from those checks. This target state is then passed to `toggle_state()`
- `toggle_state()` – this method receives the new target state from `update_state_handler()`, and studies it and the *current state* to determine:
  - If the transition should be allowed: i.e. the target is valid from the current state.
  - What actions to perform together with the transition.

  It then makes the transition, and performs the corresponding actions.

The following code demonstrates a simplification of the pattern used:[19]

```
method update_state_handler() {
  local uint64 new_state;
  switch (curr_state) {
    case STATE_A:
      if (signal_A.asserted()) {
        if (signal_B.asserted()) {
          new_state = STATE_B;
        } else {
          new_state = STATE_C;
```

[19] In addition to miscellaneous logic that was omitted, this simplification is written in DML 1.4 – in contrast, The original module uses DML 1.2. The simplified excerpts still retain the original module's use of `goto` – despite the fact `goto` is not supported in DML 1.4 – because of its impact on the code.

```
        }
        goto toggle_state;
      }
      break;
    ...
  }
  log info: "No need to update state"
  return;

toggle_state:
  local bool successful = toggle_state(new_state);
  if (successful) {
    log info: "Updated state to %s", stringify_state(new_state);
  } else {
    log error: "Can't update state to %s", stringify_state(new_state);
  }
}

method toggle_state(uint64 new_state) -> (bool) {
  switch (curr_state) {
    case STATE_A:
      switch (new_state) {
        case STATE_B:
          action_if_signal_B_asserted();
          goto success;
        case STATE_C:
          ction_if_signal_B_not_asserted();
          goto success;
        default:
          goto fail;
      }
    ...
  }

success:
  log info: "Successfully toggled state from %s to %s"
          , stringify_state(curr_state)
          , stringify_state(new_state);
  curr_state = new_state;
  return true;

fail:
  log info: "Can't toggle state from %s to %s"
          , stringify_state(curr_state)
          , stringify_state(new_state);
  return false;
}
```

The study of the current and target states in `toggle_state()` adds a significant amount of boilerplate, and is entirely redundant – the transition

is known to be valid as `update_state_handler()` calls `toggle_state()` only upon a successful context check, and what actions to be performed can also be determined within `update_state_handler()` from what context checks succeeded – it doesn't have to be driven through the study of the current and desired target state. The two methods can be merged into a single, central notification method without issue – as is, the boilerplate is more than *double* of what can be expected within typical FSM modules.