



School of Computer Science & Engineering
Trustworthy Systems Group

Verifying the seL4 Microkit

Verified libmicrokit and CapDL Mapping
Zoltan A. Kocsis, Mathieu Paturel, Isitha Subasinghe,
Trudy Weibel, Gernot Heiser
gernot@unsw.edu.au
Draft of 2023-12-21

DRAFT

Abstract

This document reports on the formal verification of the seL4 Microkit. Specifically we report on (1) the formal specification of the Microkit API, (2) the functional correctness proof of the implementation of this API in the `libmicrokit` library, and (3) the verification of a mapping of the Microkit system specification (*system description file*, SDF) to the CapDL formalism that represents access rights in an seL4-based system. Both verification steps use fully automated (push-button) techniques. All artefacts are open-sourced.

Contents

1	Introduction	3
1.1	Project background	3
1.2	seL4 Microkit Overview	5
1.2.1	Programming model	5
1.2.2	Implementation	6
1.3	Summary of outcomes	7
1.4	Synopsis of the proof work covered in the remainder	9
1.5	Repository	9
2	Global Correctness – The Long-Term Objective	10
3	Local Correctness – The Task at Hand	12
3.1	The local Microkit state machine	12
3.2	Approach taken to verify the Microkit	12
4	Microkit Local Correctness	14
4.1	Layout of formalised specification	14
4.2	Verifying implementation against spec	15
4.3	Our pipeline of verification stages	15
4.4	Verifying <code>libmicrokit</code>	17
4.4.1	Steps of the verification process	17
4.4.2	The special case of the basic Microkit library functions	19
4.4.3	Relationships of specs used in the verification process	19
4.4.4	Implementation details	20
5	Formalised Specification Underpinning Local Correctness	21
5.1	Preamble	21
5.2	Microkit abstractions – high-level specification	22
5.3	System Description File (SDF)	23
5.4	Structural arrangement of the Microkit specification	24
5.4.1	The model: state transitions	24
5.4.2	Decomposing Implementation Relation into two simpler relations	26
5.4.3	Thread-local state	27
5.4.4	Oracle	27
5.4.5	Specification of the Microkit	28
5.5	Verification condition for <code>libmicrokit</code>	29
5.6	Executing the verification steps	30
5.6.1	Proof Construction	30
5.6.2	The Microkit properties we verify	31

6	Verification of System Initialisation	32
6.1	CapDL – the key machinery	32
6.2	Approach to formalising CapDL generation	33
6.3	Choosing a CapDL loader	34
7	Formal Framework for CapDL Generation	35
7.1	Preamble	35
7.2	Abstract systems	35
7.3	Accurate implementation by capability distribution	37
8	Our New Verification Tool Gordian	39
8.1	The process steps	39
8.2	The verification condition algorithm	40
9	Existing Infrastructure Tools for Verifier Implementation	41
9.1	C Parser	41
9.2	GraphLang and SimpleExport	42
9.3	Rust CapDL loader	42
10	Worked Examples Walking Through The Verification Steps	43
11	Limitations	44
11.1	Microkit versions	44
11.2	Gap in end-to-end proof	45
11.2.1	Verifying MSpec	45
11.2.2	Haskell to SMT-LIB 2 transcription	45
11.3	Threats to validity	45
12	Achievements and Impact of Verification	47
12.1	Achievements: What we have proved	47
12.2	Impact	48

Chapter 1

Introduction

1.1 Project background

Formal verification uses mathematical methods to prove that software meets its predetermined specifications. It is one of the few techniques that can positively establish the absence of bugs in software.

The correct operation of a system depends on its underlying hardware, operating system (OS) kernel and higher-level frameworks that provide the environment in which the system executes. Consequently, bugs in the kernel or the frameworks can compromise the system's correctness and overall reliability. (Note: Throughout this report, when we use the general term “system”, we refer to a user's complete, built product at run-time; and “user” is meant in the common sense of “programmer”.)

The high-performance seL4 microkernel was the first ever general-purpose OS kernel with a formal proof of its implementation correctness, which was later extended from its source code in C to the binary code (thus taking the compiler out of the trust chain), proofs of security enforcement, and proofs of its worst-case execution time [Klein et al., 2014]. Presently, the full proof chain (including down to the binary) exists for the 32-bit Arm and 64-bit RISC-V architectures, with the implementation-correctness proof also available for the 64-bit x86 architecture. At the same time, seL4 demonstrates best-in-class performance [Mi et al., 2019], making it the ideal foundation for secure and dependable real-world systems.

However, far from being a full OS, seL4 is still only a microkernel, providing only basic mechanisms for securely multiplexing hardware [Liedtke, 1995]. Its API is policy-free and very low-level, making development of performant and correct systems on top of it costly, requiring a high level of expertise, and generally creating a high barrier to uptake.

The recently developed **seL4 Microkit** [Heiser et al., 2022] (formerly known as seL4 Core Platform) addresses this challenge by providing a small set of very simple, higher-level abstractions that are easy to use for building modular, yet still performant, systems that leverage seL4's isolation properties [Parker, 2023]. It also comes with an SDK that simplifies system generation (more on this below in the overview Section 1.2).

Microkit is still not an OS itself, but is rather a *framework* for building OS services and applications. It achieves much of its simplicity by restricting the application domain: instead of striving for generality (as the seL4 kernel does), the Microkit is designed for systems with a static architecture, i.e. ones where all components and their possible interactions (but not

necessarily the implementation of those components) are known at system-build time. These restrictions, while incompatible with desktop or cloud-hosting environments, are sufficient to support most IoT and cyberphysical systems.

The Microkit imposes an event-driven, sequential programming model on application code (also known as user code), which simplifies concurrency control and naturally leads to systems that consist of communicating sequential programs in the spirit of Hoare [1978]. This model tends to result in simpler implementations [Ousterhout, 1996]. Combined with the fact that module interfaces are enforced by verified seL4 mechanisms, that should dramatically simplify the task of verifying Microkit-based systems. However, the full benefit of verifying such systems will be realised only if the Microkit itself is verified, as errors in the Microkit will render invalid any proofs about the behaviour of applications.

Simplicity helps here as well, as the implementation of the Microkit itself is also fairly straightforward, simple enough to experiment with a more automated verification process than what is used for the seL4 kernel itself.

Verification of the seL4 kernel used interactive theorem proving in Isabelle/HOL, which supports the construction of elaborate, machine-checked proofs. While powerful in what it can prove, this manual approach is highly labour-intensive and typically requires high expertise in Formal Methods. Furthermore, it requires manual re-verification whenever the code changes.

In contrast, for verifying the Microkit, we turn to *automatic theorem provers*, specifically SMT solvers. These are tools that, once set up, can verify functional properties fully automatically, therefore frequently called “push-button verification”. This supports an agile, dynamic development style and should enable broader participation across the seL4 community.

Push-button verification was already deployed in the binary verification of seL4 [Sewell et al., 2013], and more recently has been used in functional-correctness proofs of simple operating systems [Nelson et al., 2017, 2019].

We note that while model checkers [Clarke et al., 2003; Holzmann, 1997] also have the “push-button” property, SMT solvers with a suitably chosen theory are particularly fitting for proofs of functional correctness.

However, significant challenges remain: The seL4 proofs could assume strictly sequential execution, thanks to the non-preemptible implementation of seL4 [Peters et al., 2015]. But this assumption cannot be imposed on the code that executes in user mode, such as the Microkit and any system components built on top of it, as their execution can be preempted at any time. Making verification tractable requires that this concurrency be tamed. Previous push-button verification approaches assume these challenges away, which drastically limits the guarantees that can be obtained from verification. The Microkit deals with this concurrency challenge by the fact that its functions execute atomically with respect to each other, which in turn ensures that their execution does not depend on states that can change during preemptions, except for explicitly shared memory objects (more on the notion of executing atomically in Section 5.4.3 where we introduce thread-local states).

This report outlines our verification approach and describes the steps we have carried out so far and what gaps remain outstanding. Specifically, we cover in detail the following, successful stages in the formal verification process for the seL4 Microkit framework:

A formal specification of the Microkit, or more specifically, of the Microkit library, `libmicrokit`, which is the interface between the Microkit and the seL4 kernel (it includes both functions visible to the user as well as purely internal to Microkit such as the

handler_loop, see also Figure 1.3), and of the Microkit internal PD, the monitor. We provide more detail on the structural make-up of the specification in Section 5.4.

An automated, functional correctness proof of the Microkit implementation, meaning we show that the implementations of the Microkit library plus Monitor satisfy their respective specifications (for more see Section 4.4).

A verified mapping of the Microkit system specification (written in *SDF format*) to the *CapDL* formalism [Kuz et al., 2010], which describes access rights in seL4 (as covered in Chapter 6 – 7).

We note that this mapping to CapDL will eventually connect to the verified generation of a system initialiser for the CapDL spec - we refer here to work on the *formally verified system initialiser*, called *case-init* for historical reasons, which is written in the CakeML language that has a verified compiler [Tan et al., 2016].

For the time being there remain some gaps and limitations preventing end-to-end proofs. These are discussed further in Chapter 11.

1.2 seL4 Microkit Overview

1.2.1 Programming model

The programming model presented by the Microkit API is extremely simple; all abstractions are summarised in Figure 1.1, (Section 5.2 will provide additional details).

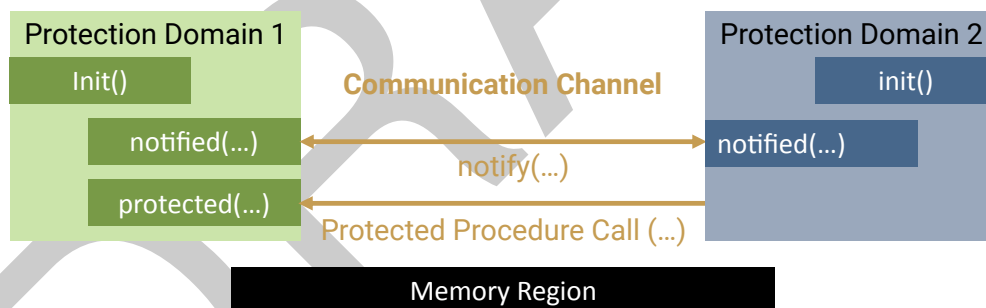


Figure 1.1: Microkit abstractions at a glance

A protection domain (PD) is the program abstraction, the primary abstraction of Microkit. It provides a simple, event-driven execution model, where the user provides the implementation of a handler, called the `notified` function, to handle interrupts (IRQs) or notifications coming from other PDs. The user must also provide a (possibly empty) implementation of the `init` function, which deals with any setup specific to the PD and is called by the system exactly once at startup time, before any of the other entry points can be invoked. The PD may also have a protected procedure `protected` implemented for when another PD invokes a PPC to it.

A PD can be a virtual machine that can run a Linux guest operating system (OS). Alternatively, a PD can be a device driver, able to receive interrupt requests (IRQs) via an implicit channel.

A communication channel (CC) connects exactly two PDs. A PD may invoke the (non-blocking) Microkit function `microkit_notify` on a CC, which will result in the

underlying system (consisting of Microkit and seL4 kernel) at some future time to invoke the notified function of the PD at the other end of that channel (the channel denotation implicitly identifies the notifying PD).

A **protected procedure call (PPC)** is the mechanism by the Microkit for implementing inter-PD function calls. In essence, a simple call is made to a user provided handler protected. We note that the Microkit library supplies functions that allow a PD of lower priority to perform a protected procedure call only to a PD with higher priority (see also the comment at the end of the first paragraph of Section 3.2).

A **memory region (MR)** is the final abstraction of the Microkit. It may be accessed by (i.e. mapped into) one or more PDs, with potentially different access rights, thus providing shared memory for communication.

This is all that is needed to build a complete system (i.e. a user's complete, built product at run-time – see also the note in Section 1.1, second paragraph).

A **Microkit-based system** is then a collection of concurrently executing PDs (modules) that communicate via shared memory, synchronise via notifications sent along channels, invoke servers via PPCs, and possibly receive IRQ notifications.

In particular, *systems built using the Microkit do not use seL4 kernel functions/services directly* (s.a. threads, device interrupts, etc.); *instead, they rely on above mentioned simple abstractions*, which are provided by the **Microkit's library**. Their composition is described in a formalism called the **system description file (SDF) format**, as set out in more detail in Section 5.3.

PDs are single-threaded.¹ The Microkit guarantees that the `init`, notified and protected functions execute atomically with respect to each other, eliminating the need for any concurrency control inside a PD – much in the spirit of the CSP model of Hoare [1978].

1.2.2 Implementation

The seL4 Microkit implementation consists of three components, which are linked to user-provided module implementations by the Microkit SDK:

- The `system_initialiser` is a special, privileged, system-provided PD, which allocates system resources. The design for static architectures makes it possible to restrict this allocation to system startup time.
- The `monitor` is a system-provided PD that runs after startup and acts as a fault handler, that is it receives notifications generated by the system if a user PD faults.
- The interface library `libmicrokit` is linked to each PD and maps the Microkit APIs to seL4 system calls. For each PD, it implements an event-handler loop `handler_loop` that waits for an event on any of the channels connected to the PD, and for each such event invokes the *user-provided* handler functions notified or protected as appropriate.

Figure 1.2 shows how these three components in conjunction with user PDs fit together as a whole seL4-based system.

The key Microkit-based functions can be grouped into three categories as per Figure 1.3:

¹Note that a multi-threaded PD can be constructed by having several PDs share the same address mapping and allocating them on different cores.

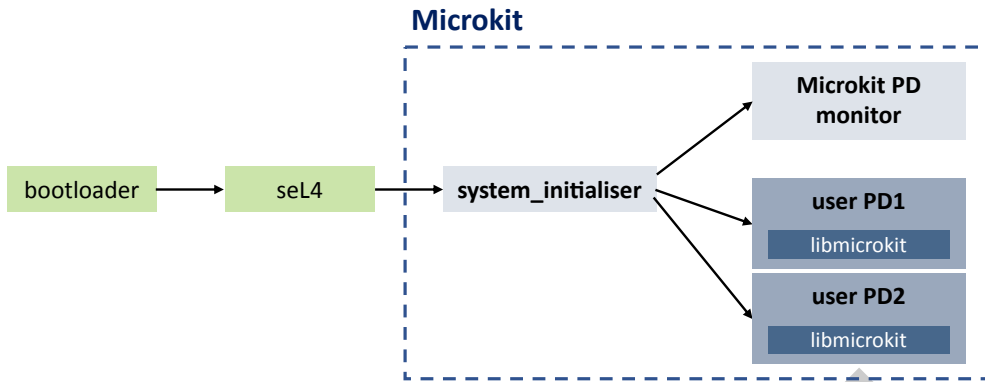


Figure 1.2: Microkit components when loaded as a whole seL4-based system.

1. in red the functions that are provided by the user (as described earlier in Section 1.2.1);
2. in blue the key functions that are provided to the user by the Microkit library (such as for notifying another PD or acknowledging an interrupt); and
3. in green the purely internal Microkit functions (most notably the event-handler loop as mentioned just above in the third bullet point).²

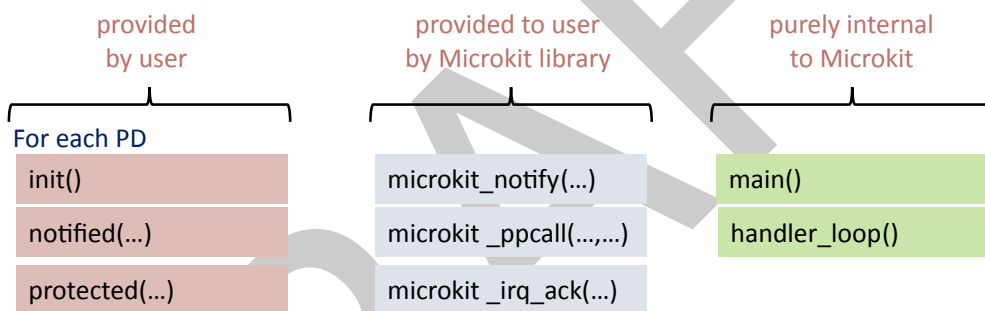


Figure 1.3: Microkit-based functions – in red: provided *by the user*; in blue: provided *to the user* by the Microkit library; in green: purely *internal* to the Microkit.

A note of practicality regarding the components: The `monitor`, while being a PD, is purely internal to the Microkit similar to other internal Microkit library functions like the `handler_loop`. Accordingly, for ease of readability, we treat from now on the Monitor as part of the Microkit library, unless explicitly stated otherwise.

1.3 Summary of outcomes

The Microkit tool (also referred to as the Microkit SDK) is a command-line Python tool that processes the user-composed Microkit SDF system description file and a set of user-provided executable files in ELF format [Wikipedia, 2001], and generates a complete system image suitable for loading by the target platform bootloader (more details about this is covered in Section 5.3).

The Microkit SDK has then been augmented to allow for incorporating/implementing the proof steps required for verifying the Microkit and its library. At the conceptual level, Figure 1.4 gives an overview of the various constituents and steps for verifying the Microkit and Microkit-based systems.

²The Microkit is a very slim wrapper of seL4: the code size of `libmicrokit` is around 300 lines of C.

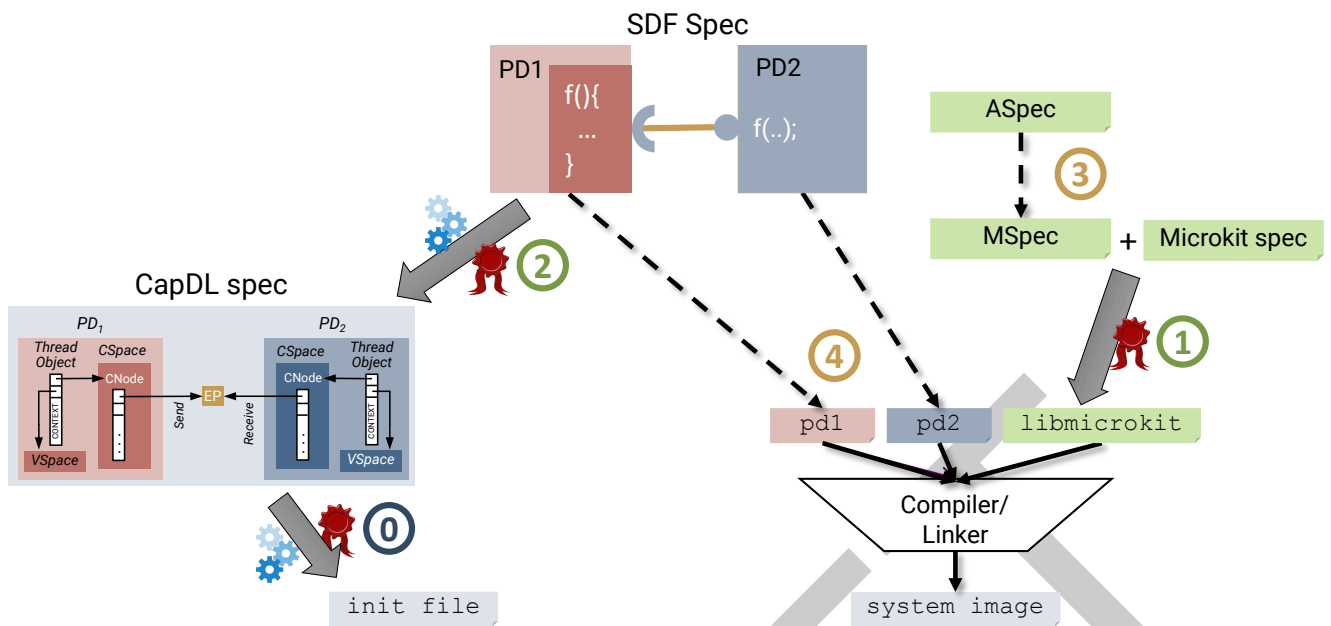


Figure 1.4: Microkit verification structure: **Proof 0 (system initialiser)** exists from prior work; **Proofs 1 (libmicrokit)** and **2 (CapDL generation)** are the result of work presented in this report; **Proof 3 (SMT-suitable abstraction)** is still required to achieve end-to-end proof (see also Section 11.2); verifying functionality of PD implementations / Mikrokit-based systems **Proof 4 (PD implementations)** is ongoing and separate endeavour.

The seL4 microkernel has an abstract specification, the **ASpec**, against which its implementation was verified, see Section 4.1 for more details. We base the verification of the Microkit on a simplified specification, **MSpec**, which is an abstraction of ASpec. MSpec describes only the seL4 functionality required by the Microkit, and is simple enough to be usable by SMT solvers. We call this abstraction step from ASpec to MSpec the **M-abstraction**. The correctness of the M-abstraction, i.e. **Proof 3**, is not yet proved, but investigations on its verification are in progress in Trustworthy Systems (see also further discussion in Section 11.2.1).

Our eventual aim is to prove functional correctness of complete Microkit-based systems (or at least their trusted computing base) by verifying PD implementations (**Proof 4**) and composing these into proofs of system-wide properties. For now our focus is on the underpinnings required for this ultimate aim: Verification of Microkit-based systems depends on correct implementation of the *Microkit itself*, i.e. libmicrokit, (**Proof 1**) as well as on correct system initialisation (**Proof 2**), both of which are achieved by the work reported here, with the latter connecting to the verified generation of a system initialiser from a CapDL spec (**Proof 0**).

Specifically, we

1. provide a specification in Haskell of the run-time components of the Microkit, i.e. libmicrokit;
2. provide an automated verification tool called Gordian, written in Python (enabling the task immediately below);
3. prove functional correctness of the libmicrokit implementation by running Gordian;
4. provide automated generation of a specification of seL4-level access rights in the

CapDL formalism (this translator is called `CapDL-tool`);³

5. provide a translation-validation framework that proves the equivalence of the generated CapDL from a given SDF spec.

We note that the last two items link, in principle, to earlier work on generating a verified system initialiser from CapDL, and would therefore guarantee correct initialisation of the Microkit-based system. However, as we expand on towards the end in Section 11.1, this link is currently limited to a fork of the seL4 kernel version, though still with a full proof-chain.

1.4 Synopsis of the proof work covered in the remainder

Figure 1.4 provides the context for the work covered in this report: We will give an account of the proofs for transitions ① and ②, namely the verification of the Microkit and of CapDL generation. The earlier chapters 2–5 will cover the verification of the run-time components, while the following chapters 6–7 will get us correctly to the starting line, i.e. initialisation. The final chapters 8–12 will then wrap up overall observations.

Chapter 2 on Global Correctness looks at the macro level of the Microkit correctness story, while Chapter 3 on Local Correctness explains how, as a first, crucial step towards it, we can hone in at the micro level. Chapter 4 then details the proof process we apply to verify `libmicrokit` and Chapter 5 discusses aspects of the formalised specification.

We then turn to the verification task of the initialisation of the Microkit when in Chapter 6 we describe the process of verifying `system_initialiser`, followed by a special focus in Chapter 7 on mathematical/formal aspects of the verification of the CapDL generation.

Chapter 8 details our bespoke verifier toolchain `Gordian` specifically developed as part of this project but designed to be generic enough to be able to be used in future for other push-button verification projects. Chapter 9 then features the various infrastructure tools we employed for these verification tasks that already exist and been employed in a verification context. Following on in Chapter 10 we walk through the whole verification process on two worked examples and show their respective artefacts that are being produced along the way.

In the final two chapters 11 and 12 we consider limitations, gaps and threats to validity, and following a summary of our achievements we point to the impact that this work has already had.

1.5 Repository

All artefacts discussed in this report, including this report itself together with the complete code, proof and documentation package, are open-sourced and can be accessed via the project website (near the bottom of that webpage).

³This translator emits an Isabelle model along with a C file to be used with the CapDL loader.

Chapter 2

Global Correctness – The Long-Term Objective

Ultimately, our goal is to prove the functional correctness of a Microkit-based system. This requires specifying the system in terms of a *global* state machine, which holds the externally visible state of the PDs, and a *trace* (temporally ordered list of the Microkit API calls and shared memory writes made by the PDs) acting as state transitions for this (global) state machine. *Correctness* can then be defined in terms of *permissible traces*.

Such a proof requires guarantees provided by the Microkit, consisting of formally verified theorems about the traces possible in any system implemented in terms of the fundamental Microkit abstractions. These guarantees in turn require a proof that the implementation of the Microkit library (`libmicrokit` including `monitor`) is correct, based on the formal specification of the kernel itself. Furthermore, the preconditions for these proofs require that `system_initialiser` correctly initialises the system.

An instructive representative of such guarantees is the following (as visualised in Figure 2.1):

Property ♣ If PD p sent a notification on a channel c , whose other end is PD q , using the `microkit_notify(c)` API call, then eventually (but no sooner than the next time PD q makes a call to receive notifications) the PD q will execute the `notified(c)` function.

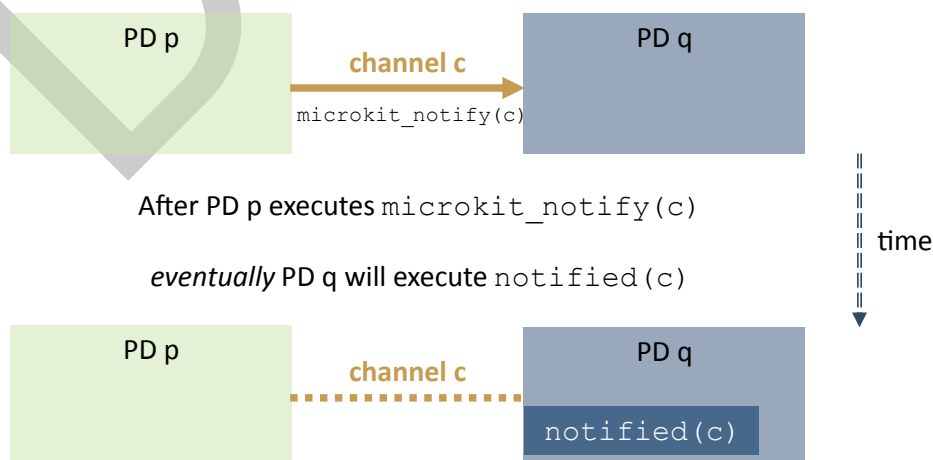


Figure 2.1: Property ♣ – a representative guarantee.

This overarching, long-term goal requires overcoming significant obstacles:

- (a) The guarantees required of the Microkit combine complicated *liveness* (finiteness of the trace) and *safety properties* (ruling out certain finite prefixes of the trace). Verification of such properties is hard with interactive theorem proving and out-of-reach for effective automated techniques – the common theories implemented in SMT solvers do not allow reasoning about unbounded traces.
- (b) One has to show that the semantics in terms of traces does not leak, in the sense that it actually captures the whole external state that a given protection domain may observe.
- (c) Property \clubsuit can hold only subject to certain scheduling restrictions. As an example, consider PDs p , q and r , where r has a higher priority than p and monopolises q with PPCs; then, any notification from p to q will never be processed, i.e. looking at Figure 2.1, `microkit_notify(c)` will be sent by p to q but q will not be able to execute the follow-up notified function because of the PPCs from the higher priority r to q . While this is a legal scenario, it can be prevented by limiting r 's time budget [Lyons et al., 2018], but this requires reasoning about scheduling behaviour, which is presently not defined in seL4's abstract specification, the ASpec.

DRAFT

Chapter 3

Local Correctness – The Task at Hand

Given the constraints as discussed in Chapter 2 above, we set about our ultimate goal of Global Correctness (proving functional correctness of a Microkit-based system as a whole) by taking a first, albeit crucial and significant step towards it: We are starting off with *Local Correctness*, where we accomplish *verification at a local level in stages* and focus on the immediate, basic work of verifying the Microkit tool itself, i.e. `libmicrokit`.¹

Furthermore, as pointed out in the introduction, we will need to deal with verifying the system as it runs, as well as with getting to a correct starting point by verifying the initialisation of the system. The latter task we will turn to in the later chapters 6 and 7.

3.1 The local Microkit state machine

We specify the Microkit API (as implemented by `libmicrokit`) in terms of a *local state machine*, also referred to as the *local Microkit state machine*, which contains only the state pertaining to the code executing in a single PD, and which describes:

1. the execution state of the PD making the Microkit API call,
2. the static configuration of the system as specified by the SDF spec, and
3. the observations that one can make during the current execution about the state of the rest of the system (such as receive calls or shared memory accesses), modeled as single-use oracles (more on this in Section 5.4.4 and Section 5.4.5).

Figure 3.1 illustrates, how closely the concepts of the local machine aligns with the model for the Microkit components as illustrated in Figure 1.2.

3.2 Approach taken to verify the Microkit

Our formal specification of `libmicrokit` describes how the various Microkit API calls made by the currently executing PD should affect its local state. In terms of the static configuration of the system and the single-use oracles, we are able to express guarantees that a correct Microkit implementation should provide. For example, we can express that a correct

¹Just as a reminder, in our context of Microkit verification we are usually considering the `monitor` to be part of the `libmicrokit` – as remarked earlier at the end of Section 1.2.2.

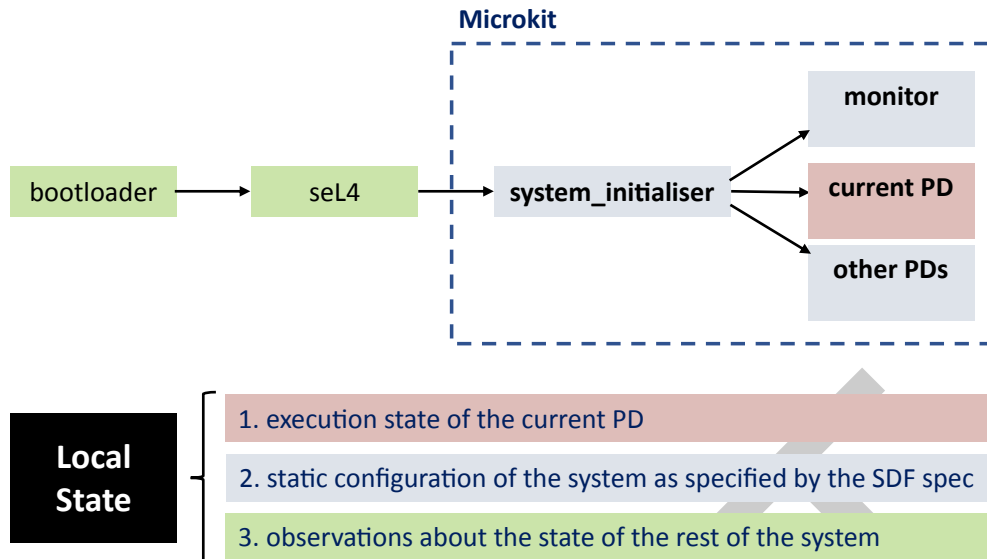


Figure 3.1: Local state capturing the three layers of the model for the Microkit components as illustrated in Figure 1.2.

implementation will not make a `microkit_notify(c)` call to a non-existent channel c , nor will a PPC be made to a PD that has lower or equal scheduling priority than the currently executing PD: We are able to guarantee the latter on verified systems even when the current implementation of the library does not programmatically enforce the restrictions.

As an instructive case, let us consider the correctness condition for the `handler_loop` (which is part of the `libmicrokit`), which executes on every PD:

Property \diamond The `handler_loop` never terminates. It will make a call to receive notifications and PPCs exactly once per iteration, and will correctly handle all responses returned by that receive call (according to the single-use oracle), including calling `notified(c)` if a notification was received on a channel c , and `protected(c, m)` if a PPC was received on channel c with argument m . Furthermore, it will not make “phantom” calls such as calling `notified(x)` on a channel x on which no notification was received.

These correctness conditions were derived with an intention behind them: Over time, we expect to make use of the facts that have been proved about the local state machine to eventually demonstrate the global correctness condition expressed in terms of traces. For example, in a global correctness argument one would appeal to the local condition Property \diamond along with a delivery guarantee coming from the kernel, to establish the global condition Property \clubsuit described in the previous chapter.

Chapter 4

Microkit Local Correctness

4.1 Layout of formalised specification

As we progress to formalise the Microkit specification, we first give a bird's eye view of the relevant features involved. Figure 4.1 captures the hierarchy of seL4 specs showing not only the proved refinement relations (see [Klein et al., 2014]) but also the currently not yet completed **step ③** of Figure 1.4.

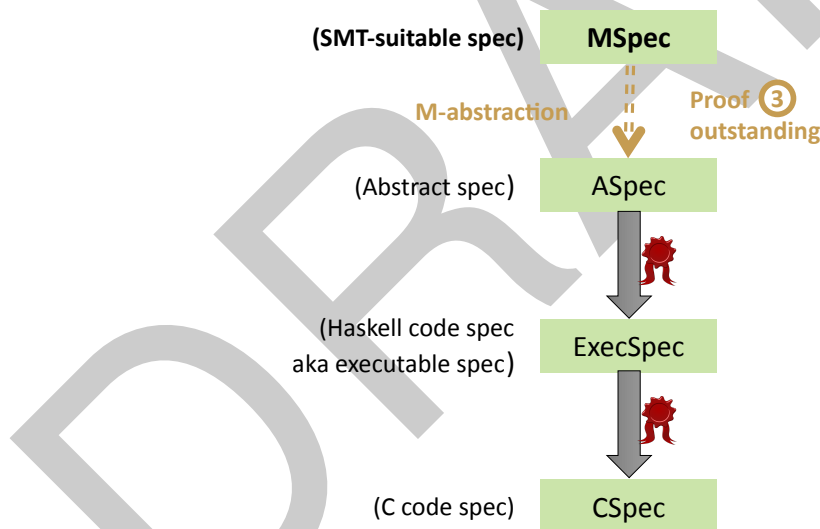


Figure 4.1: Hierarchy of seL4 specifications.

For the purpose of this report, our interest lies with Microkit relevant specs and, hence, we will focus on **MSpec**. As mentioned earlier in Section 1.3, MSpec captures what is minimally required by the Microkit from the kernel spec. We achieve this with an appropriate (minimal/SMT-suitable) abstraction of the ASpec, which we call **M-abstraction**. To clarify: MSpec is a specification of the seL4 kernel, albeit more abstract than its ASpec, it is *not* a specification of the Microkit (as for the latter, we will simply refer to it as the Microkit spec).

For the Microkit, we are starting off writing our specification in a constrained subset of the *Haskell* programming language. The *local Microkit state machine* is then defined as an algebraic data type, and each of the Microkit APIs is defined in the form of a *weakest precondition*: For each API call $f()$ and property Q we describe the weakest precondition

$F(Q)$ under which the call $f()$ either does not terminate, or it terminates and upon termination successfully establishes the condition Q .¹

The correctness guarantees are also specified. For example, the handler loop iteration is annotated with explicit pre- and postconditions ensuring that Property \diamond holds.

4.2 Verifying implementation against spec

We can now verify the `libmicrokit` implementation against its local spec. Key here is a model of the *kernel state*, and an implementation relation that relates a momentary state t of the underlying kernel to the momentary state s of the local Microkit state. We give here an initial flavour of how this is achieved. Let us consider the following, fairly high-level description:

The *Implementation Relation* $s \sim t$ defines whether the current kernel state t accurately implements the local Microkit state s .

Our definition above of the Implementation Relation allows the local correctness of an implementation to be established with a *simulation-like argument* by showing that:

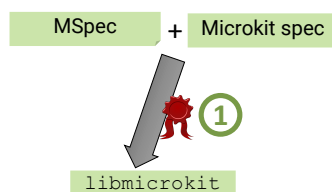
1. the `libmicrokit` implementation maintains the Implementation Relation, i.e. if $s \sim t$ holds for some Microkit state s and corresponding kernel state t , and making a Microkit API call $f()$ will leave the Microkit in some new state s' , then executing the implementation of $f()$, when starting from the kernel state t , leaves the kernel in some related state t' ;
or, in short: for all s, t , if $s \sim t$ holds then $s' \sim t'$ holds;
2. the implementation upholds all the correctness guarantees (for example, that the implemented handler loop assures the guarantee Property \diamond as defined above).

We note that \sim allows to capture functional correctness, but will not deal with properties like liveness, i.e. finiteness of traces.

Subsequent Section 5.4, especially Section 5.4.1, goes into more detail of how to exploit the Implementation Relation \sim for our local correctness proof.

4.3 Our pipeline of verification stages

Having the overarching verification picture Figure 1.4 in mind, it is **transition step ①** that is our task at hand here.



The process of verifying the Microkit implementation proceeds via multiple steps, as visualised in Figure 4.2, and comprises for the most part automated tools or, otherwise,

¹For formal reasoning purposes we allow API calls $f()$ to potentially be non-terminating, however we note that all seL4 API system calls will terminate given that their latencies are all provably bounded [Sewell et al., 2017].

essentially once-off transcription/inspection-type effort. Its various features are set out in detail in Section 4.4.1 below.

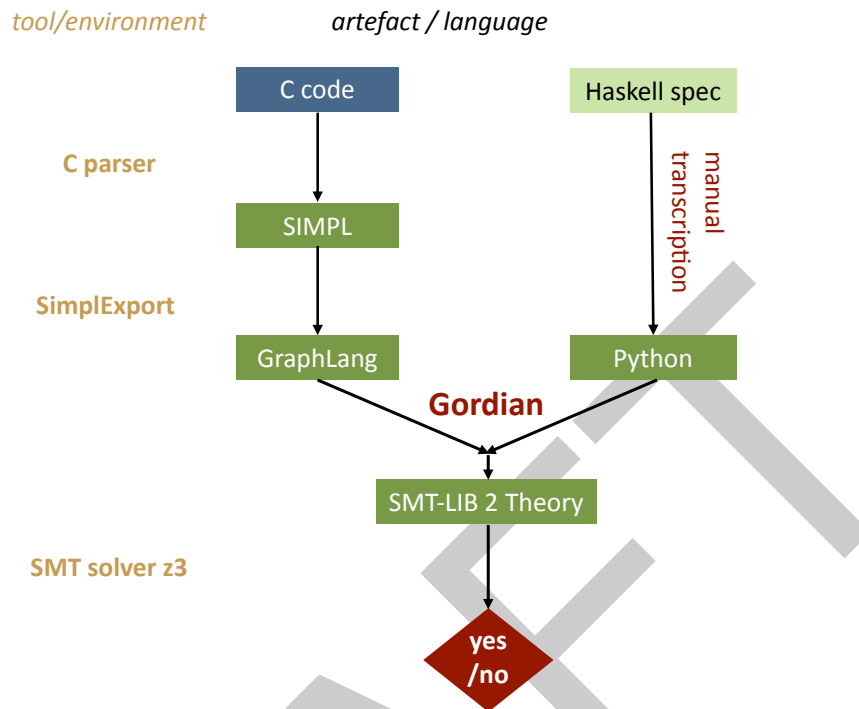


Figure 4.2: Pipeline of the Microkit verification steps - see further explanations in Section 4.4.1.

Apart from standard verification tools and environments (theorem prover *Isabelle/HOL*, programming language *SIMPL* and SMT solver *z3*), we are reusing a number of tools and libraries developed for the seL4 kernel verification effort (see Chapter 9 for more about these infrastructure tools). Such reuse is not just for convenience. It also reduces the risk of semantic mismatch, where the assumptions of one artefact in the proof chain might not be satisfied by the guarantees of the previous one. Such mismatches can be subtle and are easily obscured by unverified correspondences.

Additionally, for the functional correctness verification task of this project we developed a bespoke toolkit called *Gordian*. It is designed to bring together the source code strand with the spec strand as an automated step – Figure 4.2 nicely displaying the central role of *Gordian*. With Chapter 8 we have dedicated a whole chapter to describe the toolkit, its components and the theoretical underpinnings.

Our verification pipeline process has been designed for verification of not only the `libmicrokit` but also to allow uplifting verification to any Microkit-based programs – key here is the general C source code that the proof pipeline and its tools accept as input together with a given set of specs of `libmicrokit` functions that *Gordian* has internally available to draw upon for its proofs. We will refer to these latter functions as the *basic* Microkit library functions; they are: `microkit_notify`, `microkit_ppcall`, `microkit_irq_ack`, `seL4_Recv`, `seL4_ReplyRecv`, `seL4_SendRecv`. To ensure integrity of using these basic functions as a built-in library for *Gordian*, we need to anchor their proof independently, which we indeed have carried out as part of this project here.

4.4 Verifying libmicrokit

4.4.1 Steps of the verification process

Let us now follow Figure 4.2 and its various verification steps of the Microkit implementation. Consider a function of `libmicrokit` written in C, for example the `handler_loop`.

The C code is first processed by the same **C Parser** [Barthwal and Norrish, 2009] that is used in `seL4` kernel verification. This tool defines a semantics for a (large but well-defined) subset of the C language and translates the C source code into a semantically-equivalent program in the **SIMPL** programming language [Schirmer, 2006]. This process guarantees that we use the same C semantics as in the kernel verification.

We then perform a semantics-preserving translation of the SIMPL code into a control-flow graph in the graph language **GraphLang**², using existing **SimplExport** tools. Figure 4.3 gives an idea what these graphs look like, here in the case of the Microkit handler loop.

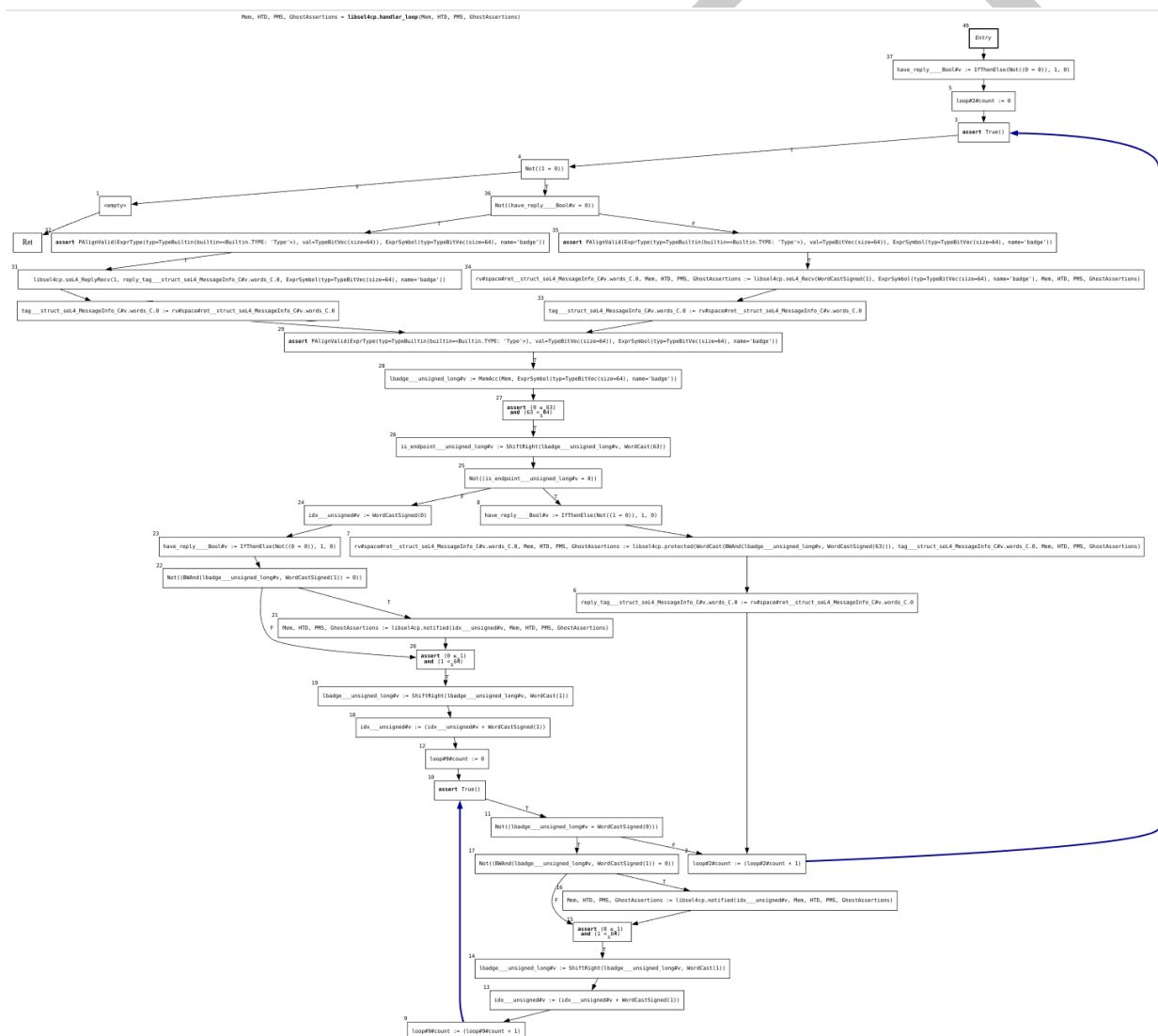


Figure 4.3: This control-flow graph is generated by the graph language tool `GraphLang` for the `libmicrokit` function `handler_loop`. (Here in the graph, loop edges are marked in blue, whereas `Err` nodes that result from false branches of `assert` statements are left out.)

²`GraphLang` was formerly known as `SydTV-GL`, Sydney Translation Validation Graph Language.

GraphLang is a common intermediate language, able to represent essentially arbitrary, unstructured control-flow. It is already used in the binary verification of the seL4 kernel [Sewell et al., 2013]. (This same tool-chain will be a natural candidate for the eventual proof of the *binary libmicrokit*.)

Alongside an implementation of Microkit in C we have generated a formal specification for it in **Haskell**. This spec is then *manually transcribed* into **Python**.

For the next step **we have developed a new, tailor-made, automated tool**, called **Gordian**. It first takes the above Python description and generates a spec in **SMT-LIB 2**. Gordian then verifies the given GraphLang graph against the generated SMT-LIB 2 spec. It does this by annotating the graph with the specification, from which it generates a single, logical verification condition, using a variant of the weakest-precondition calculus for unstructured programs of [Barnett and Leino, 2005] – we will refer to this as the *Barnett-Leino-variant algorithm*. Furthermore, when Gordian comes across references to basic Microkit library functions it directly relies on their respective (pre-/post-condition) specs – see Section 4.4.2 how we avoid circularity and properly anchor verification of these basic functions. Figure 4.2 highlights the key role that our Gordian tool plays, Chapter 8 details its particular features and capabilities.

The **verification condition** thus produced is a formula in a standard **SMT-LIB 2 theory**,³ meaning we can employ for our reasoning the common and standardised input and output language for SMT solvers. This verification condition is then passed to the SMT solver **z3** [de Moura and Bjørner, 2008], which either proves the condition or provides a counter-model.

Thus, in summary, if the verification condition is proved, it means that the function’s implementation in C satisfies its specification, i.e. that we have functional correctness.⁴

Additionally, we aim to ensure behavioural properties of the kind of *absence of common programming errors*, such as:

- No null pointer dereferences and inadmissible memory accesses;
- No incorrect uses of dynamic memory during program operation (no ill-typed nor dangling pointers, no out-of-bounds errors, etc.);
- No arithmetic overflows and exceptions (no signed integer overflows, no division-by-zero, no invalid bit shifts, no invalid conversions, etc.);
- No other undefined behavior (not trying to use values of uninitialised local variables, etc.).

We effectively achieve this “good behaviour” practically for free as we establish that the C Parser semantics is met as part of our functional correctness process.

At this point it is important to note that for our Microkit library functions, non-termination is always an allowed behavior. For example, the handler loop is specified to never terminate, and the verifier explicitly confirms this property.

³In our case, we have mostly been using the standard SMT-LIB 2 theory QF_ABV (quantifier-free arrays with fixed size bit vectors theory), or then the standard theories for quantifiers and algebraic data types.

⁴At this stage, the weakness of our verification process here is the translation of the function spec from Haskell to Python to SMT-LIB 2: while straight forward, it is manual and not formally proved.

4.4.2 The special case of the basic Microkit library functions

At the end of Section 4.3 we introduced the notion of *basic* Microkit library functions, {`microkit_notify`, `microkit_ppcall`, `microkit_irq_ack`, `seL4_Recv`, `seL4_ReplyRecv`, `seL4_SendRecv`}. These functions are verified using a slightly simpler approach than that outlined above in Section 4.4.1. As a reminder, Gordian’s full power includes relying on correctness of the basic functions. It is this feature that naturally must be turned off when submitting one of these basic Microkit library functions itself for Gordian verification. This effort of special verification treatment is once-off and only for the six basic functions. Thereafter, the verification process as laid out in Section 4.4.1 is then applicable in the universality as intended for general Microkit-based programs and systems.

4.4.3 Relationships of specs used in the verification process

With Figure 4.4 we depict how the various specifications employed in the `libmicrokit` verification process are connected, and how much or which portion of this has been accomplished as part of our project here.

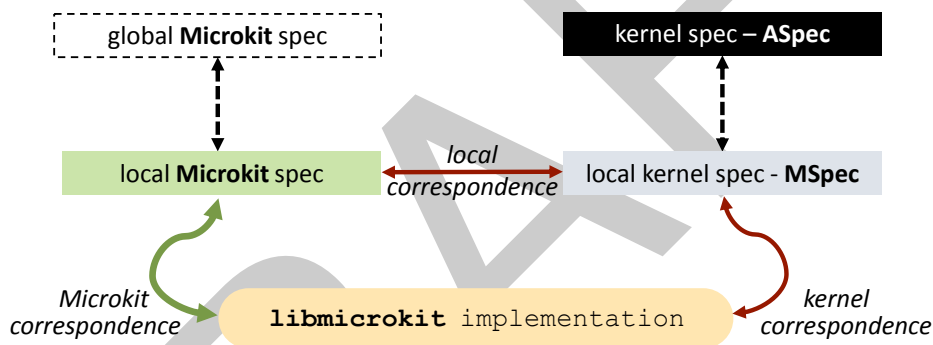


Figure 4.4: Relationships between the specifications for `libmicrokit` verification. Green arrow: formally verified correspondence as part of this project; dark red arrows: correspondences with proofs near completion; dashed arrows: correspondences conceptualised but unverified.

Specification – rectangular box

These specification relationships involve several specs that are also discussed further in this report, mostly in Section 5.4.

light green/grey box: these specs are fully developed as part of this project:

- “local Microkit spec”,
- “local kernel spec” (“MSpec”);

black box: “kernel spec” (“ASpec”) is the pre-existing formalisation against which the kernel had been verified [Klein et al., 2014];

dashed white box: “global Microkit spec” is only conceptually formulated – as discussed in Chapter 2.

Correspondence – arrow

Similarly for the correspondences, they are developed to various degrees and are discussed further also in Section 5.4.

green arrow: “Microkit correspondence” is formally verified as part of this project using our tool Gordian – additionally see Section 5.4.5;

dark red arrows: both “kernel correspondence” and “local correspondence” have their formal verification near completion;

dashed arrows: these correspondences have been conceptualised but remain unverified/unimplemented - see also comment to proof step ③ in Figure 1.4

4.4.4 Implementation details

The GraphLang export of `libmicrokit` consists of 3,540 lines of code. The above proof chain with the Gordian tool is able to verify the functional correctness of `libmicrokit`. Using the z3 SMT solver as the main backend, the verification takes about 20 seconds on a desktop computer.

The proof is fully automated: no manual proof effort is required once the SMT-LIB 2 spec is written (the latter is, at this stage, obtained through direct, manual transcription from Haskell).

DRAFT

Chapter 5

Formalised Specification Underpinning Local Correctness

5.1 Preamble

We are now adding some more flesh to the overview spec chart Figure 4.4. The augmented Figure 5.1 now captures details about the form/kind of the various specifications, what functions are formally specified, and generally what the interdependencies are of the various specifications and semantics of the Microkit components and their seL4 kernel context when ultimately proving libmicrokit.

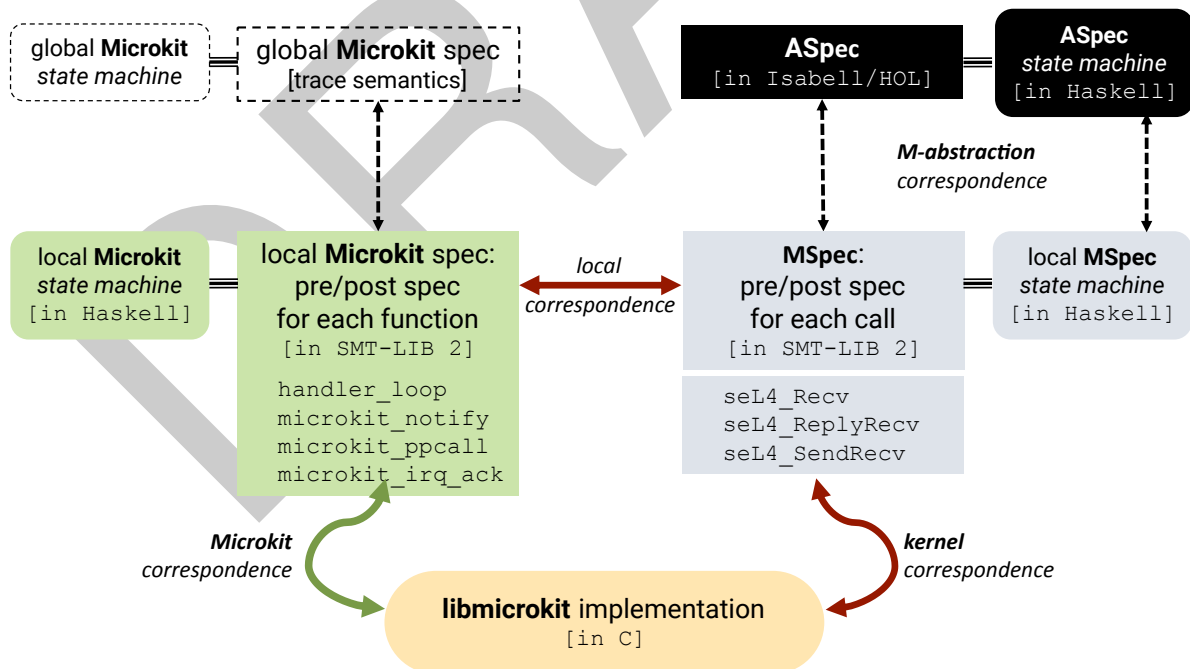


Figure 5.1: Interdependencies of the specification elements for the Microkit verification. The boxes indicate the following: specs are drawn rectangular with their corresponding state machines shown out to the sides with small rounded corners. Spec colour in black: prior formally verified component of the kernel seL4; in light green: formally verified component as part of this project; in light grey: formally specified but not (yet) verified component; in white: only conceptually, but not formally specified component.

In the following we present the formalism for the specifications, the semantics and the relationships between them. We begin with the high-level spec, which also allows us to describe the general approach of verifying the Microkit. We then provide insight into formal details of the specifications to indicate how we achieve strict verification proofs.

By way of setting the scene, let us recap our overarching goal: to make seL4 easier to deploy in security-critical scenarios, while raising the assurance levels of both `libmicrokit` itself and any operating system developed using this tool kit. As mentioned in the introduction Section 1.1, unlike the *interactive* theorem-proving approach that was required for the seL4 kernel, here we have set our eyes on using the *automated* theorem-proving approach with SMT solvers. While the latter, once set up correctly, provides a sustainable assurance model (via *automatic* re-verification when maintaining underlying source code), getting to that point is still challenging for non-trivial systems, and frequently relies on strong assumptions about the environment of the code to be verified.

In contrast, here for our project we assume no more than the *seL4 proof guarantees*, that is to merely rely on the results of [Klein et al., 2014], the seminal work about our verified seL4 kernel as introduced in Section 1.1.

5.2 Microkit abstractions – high-level specification

The Microkit is a minimal operating system framework with an SDK built to run on the proved secure, safe, and reliable seL4 microkernel. The kernel itself provides a small number of services for implementing systems, such as capability-based access-control primitives, device interrupts, endpoints for message passing, virtual address spaces, threads and scheduling contexts. However, *systems built using the Microkit do not use these kernel services directly. Instead, they rely on four simple abstractions provided by the Microkit's library.*

The four abstractions are PD, CC, PPC and MR, as introduced in Section 1.2 and Figure 1.2 when we conveyed how a system implemented with the Microkit is composed of only four types of first-class objects with explicitly stated relationships between them.

These Microkit abstractions were chosen to closely mirror objects of the seL4 kernel, and were deliberately designed in such a way that they lend themselves to very efficient implementation in terms of the services provided by the kernel, while ruling out common design issues that arise when seL4 abstractions are used directly in certain inappropriate ways.

Further to their initial definition we note a few additional observations about these four core objects:

Protection domain (PD): The PD is the process abstraction of the Microkit: It encapsulates a thread (executing a copy of the `libmicrokit`'s event-handler loop and linked to user-provided code), a virtual address space and a capability space, all set out in a certain, well-defined manner. Each PD also has scheduling information, including a priority, associated with it.

Communication channel (CC): Two PDs may have a channel between them: `libmicrokit` provides functions for sending and receiving *asynchronous* signals across channels. On the implementation side, the existence of a channel between two PDs asserts the presence of certain capabilities in the capability spaces of the associated PDs.

Receiving and acknowledging device **interrupts (IRQs)** is also implemented using a slight variant of the channel concept.

Protected procedure call (PPC): PPC is an operation on a channel that invokes the protected function in the PD across the channel, which executes *synchronously* with the caller, i.e. the underlying system will cause the protected function to execute, with any return value from the protected function being returned to the caller before the caller can continue. The protected entrypoint is optional, meaning that only some PDs, informally called “servers”, may be invoked via a PPC – PPC is the Microkit’s equivalent of a system call in a monolithic OS. The system requires (but does not enforce programmatically) that a PPC can go only to a higher-priority PD (thus preventing deadlock).

Memory region (MR): It represents a known, contiguous range of physical memory. A memory region can be mapped into the virtual address space of (i.e. accessed by) one or more PDs with possibly different privileges. As with channels, on the implementation side the existence of a mapping between a memory region and a PD asserts the presence of certain capabilities in the capability space of the PD, and facts about the layout of the virtual address space of the PD.

In Section 7.2 we will do a deep dive and go full-out formal on specs and include an account of which capabilities are to be provided by which abstractions.

5.3 System Description File (SDF)

On the back of these abstractions we now target the formal specifications of the two main components of the Microkit, the `libmicrokit` library and the `monitor`. For that purpose, let us start with SDF, the composition mechanics of the Microkit available to the user.

A user who wants to build a Microkit-based system, first compiles and links each task, providing the notified, `init` and possibly protected functions, into a separate ELF file.

Furthermore, an XML **System Description File (SDF)** describes which PDs will be present in the system, which channels will be present between them, and which PPCs will be allowed, etc.¹ See Listing 5.1 for a simple, instructive example of an SDF specification.

These user ELF files together with the SDF completely specify the user’s system, and in conjunction with the Microkit SDK are sufficient to build the system image (using the user’s preferred build tools), as shown in Figure 5.2.

A particular feature of a Microkit-based system is its static structure in the following sense: Its properties as described in the SDF spec (such as the number of PDs, the channels between them, the mappings and permissions of the memory regions, which PDs allow PPCs, etc.) never change during the execution of the system. Therefore, in the formal model of the Microkit framework, the data specified in the SDF spec is collectively referred to as the **Microkit Invariants** (just below they will play a key role in the design of the verification approach).

¹The description of the SDF format can be found in the Microkit manual.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <system>
3   <protection_domain name="server"
4                       priority="200"
5                       pp="true">
6     <program_image path="server.elf" />
7   </protection_domain>
8   <protection_domain name="client"
9                       priority="50">
10    <program_image path="client.elf" />
11  </protection_domain>
12  <channel>
13    <end pd="server" id="1" />
14    <end pd="client" id="1" />
15  </channel>
16 </system>

```

Listing 5.1: SDF specification of a simple system consisting of two PDs connected by a channel, where PD “server” has a PPC while PD “client” does not, and the channel identifier is “1”.

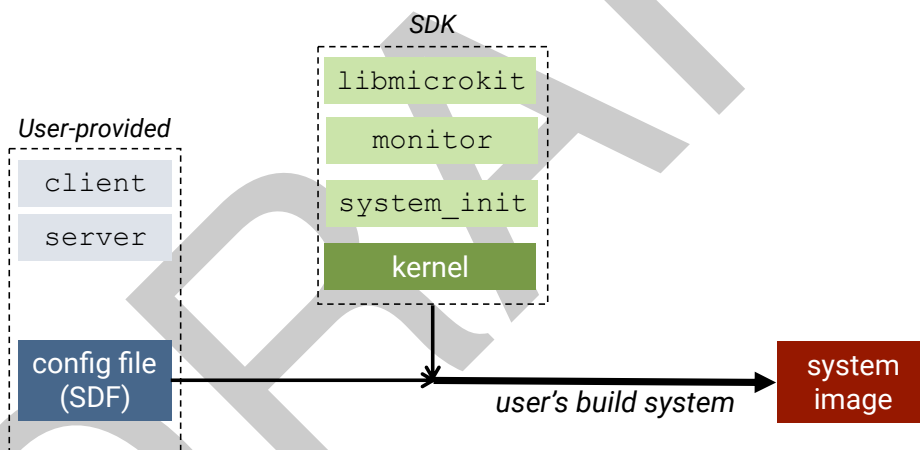


Figure 5.2: Completely built system using the Microkit: The user provides implementation of PD functions (in light grey) and a config file SDF (in darker grey); the SDK (in green) organises the Microkit library, monitor and initialisation functions plus kernel calls; the user’s own build system then constructs the final, complete system image.

5.4 Structural arrangement of the Microkit specification

5.4.1 The model: state transitions

Verifying the implementation of the Microkit library against its local specification requires several ingredients. In fact, we have already come across them in the earlier structural spec diagram Figure 5.1 but now we provide definitions and put them in context.

The main ingredients are:

Microkit State – whereby the Microkit is modelled as a state transition system, and phrased purely in terms of the four Microkit abstractions, as presented earlier in Section 1.2 and 5.2, with the corresponding spec called *local Microkit spec*.

Kernel State – whereby on the implementation side its model is again a state transition system, and which is phrased in terms of abstractions that the kernel provides.²

Implementation Relation \sim (aka **local correspondence**) – a relation telling us which kernel state (X) *is consistent with* which Microkit state (x), usually written as $x \sim X$.

We can now **specify each function** of `libmicrokit` by asserting the preconditions for invoking the function, and by defining the state transition the function causes in the Microkit state. In standard manner, we declare:

The Microkit state transition f **is correctly implemented** by an implementation program F precisely if whenever a Microkit state x satisfies the precondition of f and is related to a kernel state X , then the state $f(x)$ of the Microkit after invoking f satisfies the postcondition of f and is still related to the state $F(X)$ of the kernel obtained by executing F when starting from the state X . In short (whereby \supseteq indicates *is correctly implemented by* or *is refined by*):

$$\begin{aligned} f \supseteq F \\ \text{iff} \\ \forall x, X. \text{pre}_f(x) \wedge x \sim X \implies \text{post}_f(f(x)) \wedge f(x) \sim F(X) \end{aligned} \quad (5.1)$$

whereby Equation 5.1 is referred to as the **verification condition**.

Given that, as often is the case and particularly so with our project and its `libmicrokit` functions, the implementation programs come adorned with pre- and postconditions and can be relational in nature rather than strictly functional, we in fact work with the following **modified, more explicit formula as our verification condition**:

$$\forall x, X. \text{pre}_f(x) \wedge x \sim X \implies \text{post}_f(x, f(x)) \wedge \text{pre}_F(X) \wedge \text{post}_F(X, F(X)) \wedge f(x) \sim F(X) \quad (5.2)$$

Thus, the verification conditions will have to establish that the implementation relation \sim between the Microkit and kernel states is preserved when accounting for satisfaction of the pre- and postconditions.

A side remark

For our implementation relation \sim we are free to choose whatever we deem suitable. However:

- If \sim captures only few constraints, less work is required for the proofs but we are left with only weak expressiveness. (In this case, too many kernel states are related to a Microkit state, and thus, many post-program kernel states will be related to the post-transition Microkit state.)
If \sim is overreaching, it can all get too complicated and unduly detailed.
- As we will see in Section 5.4.2, to suit our specific needs we want to ensure that the chosen implementation relation \sim maintains the invariants as derived from SDF specs, in the sense that if $\text{inv}(x)$ and $x \sim X$ then $\text{inv}(X)$.
- In short, choosing the right balance and the right kind of \sim constraints is a skilled art when setting up a verification configuration.

²This explains the name “kernel state” for the state of the implemented Microkit; we note that this state may also be referred to as “implementation state” or “thread state”.

5.4.2 Decomposing Implementation Relation into two simpler relations

Let us first introduce two further concepts that are needed for the decomposition and will help illuminate the verification task more generally.

Microkit Invariants describe the static properties of a Microkit-based system (and as per last paragraph of Section 5.3 are captured with the user-provided SDF config file): the protection domains, channels, protected procedure calls, memory regions, interrupts. These properties can differ between two systems, but are fixed for any one single system (in particular under state transitions).

Microkit Dynamic State contains that portion of the state that may change with each state transition; in particular, it reflects relevant pieces of the kernel state (that is, the Microkit Dynamic State includes parts that correspond exactly to a piece of the kernel state that is relevant to the Microkit).

The Implementation Relation \sim can then be decomposed into two simpler relations as follows.

Instead of $x \sim X$, i.e. that a given *kernel state* X is consistent with the *Microkit state* x , we now require:

Microkit Invariant preservation – that a given kernel state *satisfies the Microkit Invariants*.

and

Microkit Dynamic State relation – that a given kernel state *is consistent with the Microkit Dynamic State*.

Figure 5.3 at the end of this chapter provides a visualisation of how the states and these two simpler relations are coupled with each other.

In Section 5.5 we will make use of the decomposition when creating the proof condition for the correctness of the Microkit library implementation. But for now, we make the following observations of what the split-up into these two simple, but illuminating relations achieves:

- The Microkit Invariants only constrain what kernel objects the system will have, and how capabilities to those objects will be distributed. Consequently, the relation between kernel state and Microkit Invariants is completely determined by the SDF-CapDL mapping (as set out later in Chapter 7) and will therefore only need to be shown to be established at time of initialisation.
- In our proof chain, while we *assume* that all implementations (i.e. kernel states) preserve the Microkit Invariants, we do *show*, using that assumption, that the implementations (i.e. kernel states) also preserve the Microkit Dynamic State relation.
- Assuming Microkit Invariant preservation is justified and safe: We made it a proof obligation for verified system initialisation that such initialisers will initialise the Microkit into a state consistent with the Microkit Invariants. In the introduction to Chapter 6 we expand on how this is achieved.
- Furthermore, assuming Microkit Invariants are established (as per above observation), Section 5.5 then establishes that no kernel call made by any PD will ever put the system into a state that is not consistent with the Microkit Invariants. And thus, the Microkit Invariants live up to their name and will hold and no user implementation can fail to preserve them.

5.4.3 Thread-local state

Even though the Microkit (or a user's Microkit-based system) is (part of) an operating system, it runs in user mode on top of the kernel. The kernel itself is single-threaded, but PDs run concurrently with each other and the kernel.

In principle, it would be possible to develop a Microkit State structure that represents the entire static and dynamic state of a Microkit-based system. However, this would be both overkill for our present purposes (since we only intend to prove properties about single instances of `libmicrokit`, and each PD runs in a single thread), and would make automated reasoning difficult or impossible. The latter is essentially driven by the following understanding. We wish to reason about our code in a Hoare-style precondition/postcondition framework, and extension of Hoare logic to concurrent programs is not an entirely solved problem: Even if there were tried-and-true methods for verification condition generation in such frameworks, the combinatorial explosion caused by having to analyse preemption, order of execution, etc. threatens to exceed the capabilities of our SMT solvers.

Localising states

Consequently, our formal construction of the Microkit Dynamic State will not include the whole state of the Microkit, but rather only the state *pertaining to the current protection domain*. Similarly, the kernel state will not include the whole state of the system, but only the state that is *relevant to the implementation* of `libmicrokit` inside the currently running thread.

To make this precise, we introduce the notion of *thread-local state*. Consider the abstract specification of the seL4 kernel, the ASpec, and its abstract, global kernel state S , a datum d in S and a thread t . We then define:

The datum d is part of the **thread-local state** of the thread t if the following holds:

In any abstract, global kernel state reachable from S , the datum d cannot change unless the thread t makes a kernel call.³

In particular, every sequence of kernel calls that changes the datum d includes at least one call made by the thread t .

While the kernel state of a thread t is required to be a subset of the state of the entire system,⁴ it can be, and is desirably so, a suitable, small such subset; and it consists of only thread-local state of t , i.e. “kernel state of t ” = “thread-local state of t ”. Also, just as the full kernel state is captured with seL4's ASpec, we use the name **MSpec** for the specification that corresponds to that suitable subset, and the formation of MSpec from ASpec we call **M-abstraction**.

5.4.4 Oracle

Notification delivery and PPCs depend fundamentally on concurrency, in the sense that the received notifications and procedure calls depend on what the other protection domains chose to do. We can model the concurrency-dependent operations using *oracles*, namely variables whose values tell us what the concurrency-dependent calls *will* return in the future, when the current thread chooses to make a concurrency-dependent call.

³This is sometimes also known as *d is owned by t*.

⁴An analogous case is for instance a value of type `KernelState` in `SEL4.Model.StateData` of `l4v`

Normally, such oracles could be represented as lists or more advanced data structures, but since our verifier encodes properties in SMT for automated verification, we had to choose a representation that avoids unbounded data structures and recursion. We opted to model the oracles as *single-use*.

Intuitively, single-use oracles are *filled* by the global state machine, and *consumed* by the local one. This allows the former to summarise and communicate complex information to the latter (notably, information only available in the traces).⁵ For example, when a PD's handler loop receives an event, it consumes the receive oracle and deduces the return value from it. Beforehand, the global state machine had filled the oracle appropriately by observing the trace of the system as a whole.

More technically, at the beginning of each execution iteration, the oracles become available. The availability of the corresponding oracle becomes a precondition of each kernel call, so a kernel call cannot be made unless the oracle is available. Similarly, the unavailability of the corresponding oracle becomes a postcondition, so the oracle becomes invalidated, i.e. unavailable, after a kernel call which otherwise might have affected the return value of the next call.⁶

5.4.5 Specification of the Microkit

With the concepts from earlier this section, we now declare the **full implementation state** of a thread and the associated PD to consist of the following:

- the Microkit Invariants,
- the Microkit Dynamic State of the PD, including the states of the Microkit oracles,
- the thread-local kernel state of the thread, and
- the states of the kernel call oracles.

This then leads naturally to define the **Microkit specification** to consist of

- pre- and postconditions for the Microkit state transitions, phrased in terms of the Microkit Invariants and the Microkit Dynamic State, and
- pre- and postconditions for the kernel state transitions, phrased in terms of the full implementation states (both Microkit and kernel states).

Specification in Haskell

Our specification of `libmicrokit` is represented as a Haskell program following the structure set out above in this section. The Haskell program defines data structures for the Microkit Invariants and the Microkit Dynamic, the kernel and the oracle states, and defines pre- and postconditions for the state transitions (usually in weakest-precondition form).

⁵From the perspective of the local state machine, oracles predict the future, hence the name.

⁶The oracle becoming unavailable once used is the reason it is referred to as "single-use".

5.5 Verification condition for libmicrokit

Understanding the structure and detail of the specification of the Microkit allows us to get to the core business: formulating the verification condition that is required for verifying the Microkit library.

As a reminder, in Section 5.4.1 we laid out what it means that a Microkit state transition f is correctly implemented by an implementation F in terms of kernel state transitions:

$$f \text{ is correctly implemented by } F \text{ iff } \forall x, X. \text{pre}_f(x) \wedge x \sim X \implies \text{post}_f(f(x)) \wedge f(x) \sim F(X)$$

That is, the verification conditions will have to account for satisfaction of the pre- and postconditions, and establish that the implementation relation \sim between the Microkit and kernel states is preserved.

Given the decomposition of the Implementation Relation \sim into Microkit Invariants and Microkit Dynamic State relation, and that Microkit Invariants only need to be established at initialisation (see first observation in Section 5.4.2), we are turning our attention to the Microkit Dynamic State relation. We are verifying the Microkit Dynamic State relation preservation stepwise from the local to comprising the Microkit as a whole.

- Since the thread-local state can change only when t is executing, we can reason about the parts of the implementation that involve only the thread-local state using purely sequential techniques.
- This naturally calls for a very localised obligation, namely that none of the kernel calls made by t cause the thread-local state to cease being thread-local.

The proof of this property relies, in a first instance, on the fact that with seL4 we use a verified kernel, and practically, it relies on not using the kernel's full specification (ASpec), but rather using the small subset (MSpec).

- We then verify that kernel calls made by any thread cannot cause the thread-local state in our suitable subset/M-abstraction to cease being thread-local.
- The above steps allow constructing the proof of preserving the Microkit Dynamic State relation.

FYI, Zoltan's original text from report 3.1.2 Jun 2022, section 2.2.2

The kernel state of a thread t is required to be a projection (subset) of the state of the entire system (analogous to e.g. a value of type `KernelState` in `SEL4.Model.StateData` of `l4v`) consisting only of thread-local state of t . Since the thread-local state can only change when t is executing, we can reason about the parts of the implementation that involve only the thread-local state using purely sequential techniques. Naturally, this creates an obligation for us to show that none of the kernel calls made by t cause the thread-local state to cease being thread-local: we eventually aim to verify that kernel calls made by any thread cannot cause the thread-local state in our projection to cease being thread-local as part of the proof that the *Platform Invariants are preserved*.

To summarise, we have achieved the following:

- As observed in Section 5.4.2, once the Microkit Invariants are established at the time of initialisation (see Chapter 6) we can then take them as being preserved at runtime as well.
- Above, we have set out the proof of preserving the Microkit Dynamic State relation.

- Since these two relations make up the Implementation Relation \sim , the preservation of \sim is therefore ensured.
- With this in mind, we apply this preservation proof to all the Microkit state transitions and their implementations. Specifically, our SMT-based verifier establish that the *implementations of the libmicrokit handler loop and all user-facing libmicrokit calls are correct.*
- In short, this results in a correctness proof of the Microkit implementation.

In the diagram Figure 5.3 we draw out the relations that are preserved between the Microkit and the kernel contexts, and mark how the full abstract kernel state (ASpec) projects into the kernel thread-local state (what we name M-abstraction). In particular, it shows the two simpler relations that the Implementation Relation \sim is decomposed into, namely “*satisfies the Microkit Invariants*” and “*is consistent with the Microkit Dynamic State*” (as defined in Section 5.4.2).

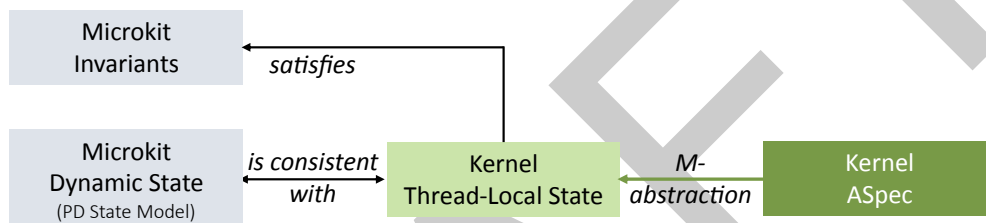


Figure 5.3: Overview of the preserved relations between the Microkit (light grey) and the kernel (light green) contexts, and the relationship between the Kernel Thread-Local State and the full abstract kernel state (dark green).

Isitha/Mathieu: In the Example, is it safe to call the function “*microkit_receive*” or do I need to use its actual name?

5.6 Executing the verification steps

5.6.1 Proof Construction

To verify the implementation of the Microkit, we have to transform each function into a logical verification condition which can be handed to an SMT solver. This verification condition should be the most general precondition relative to the function’s specification.

The proof linchpin then is:

- If an SMT solver can confirm that the *negation of the verification condition is unsatisfiable*, we can consider the function *correctly* implemented.

The steps to get there are:

1. Each C function in the implementation is annotated with its formalised specification.
2. The C source code is translated into SIMPL using the C Parser.
3. The SIMPL artefact is translated into the graph language GraphLang code via Simplexport.

4. Verification conditions are generated by the Python implementation of the Barnett-Leino-variant algorithm (as introduced earlier in Section 4.4.1) and the invariant-finding heuristics.
5. The verification conditions are checked by multiple SMT solvers.

5.6.2 The Microkit properties we verify

When verifying the Microkit we verify a range of properties about the implementation of the Microkit:

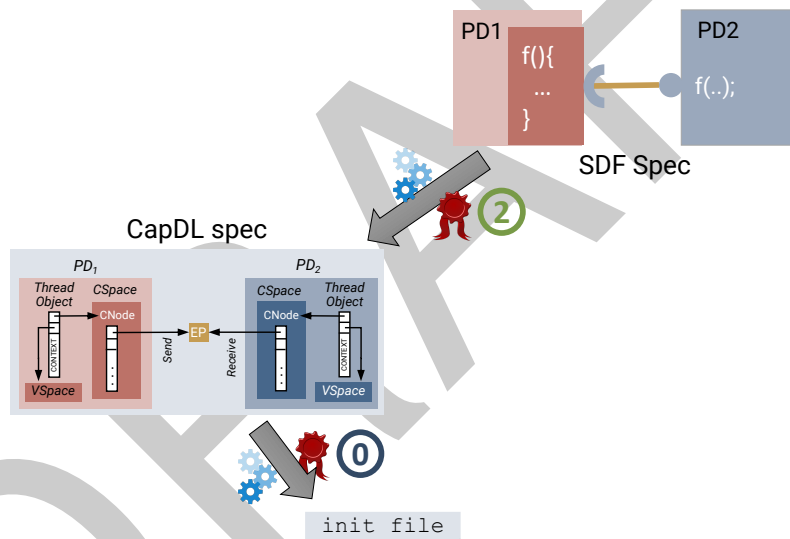
1. The code in `libmicrokit` and the `monitor` does not fail, and no undefined behavior is encountered according to the C semantics induced by the C Parser. Among other things, this means that assertions never fail, no null pointer dereferences or out-of-bounds accesses are performed outside of user code.
2. All user-facing `libmicrokit` calls terminate, even calls with misinformed input.
3. The handler loop satisfies its specification. The handler loop never terminates. Once an iteration of the loop is completed, `libmicrokit` code will have either received all pending channel notifications and invoked the corresponding channel's user-supplied notified method exactly once, or the code will have handled a protected procedure call.

DRAFT

Chapter 6

Verification of System Initialisation

After having addressed the correctness of the Microkit at run-time, we now turn our attention to a *correct initialisation* of a Microkit-based system as captured by the left-hand side of Figure 1.4, specifically the **new proof step ②** combined with the subsequent, existing proof step ①.



In the prior chapter towards the end of Section 5.4.2, we proved functional correctness of the Microkit under some assumptions, namely that the Microkit-based system is initialised into a state consistent with the Microkit Invariants. It is in this chapter, that we now show that, indeed, this assumption holds.

We achieve this by linking the user system spec as defined in the SDF config file with a system capability distribution, in our case defined in CapDL, and show that this link is a faithful representation. For this purpose, we can choose from a few suitable system initialisers, amongst them one formally verified one.

6.1 CapDL – the key machinery

CapDL is the language used to describe access rights in seL4-based systems [Kuz et al., 2010]. CapDL specifications can be used to track which objects and entities have access to which seL4 capabilities, and to provide complete descriptions of the capability distribution in a

system running on the seL4 kernel. All this makes CapDL a powerful and versatile tool for managing seL4-based systems.

There are several tools which can initialise an seL4-based system into a state that is described by a given CapDL distribution spec. These include:

- the original `capdl-loader`, written in C and maintained by Trustworthy Systems;
- a new `rust-capdl-loader` [Spinale, 2023]; and
- a *formally verified system initialiser* called `case-init`, which is written in the CakeML language, which, in turn, has a verified compiler [Tan et al., 2016].

We augment the Microkit SDK with functionality to automatically generate CapDL language output corresponding to the system specification as captured in the SDF config file (see also Figure 5.2). This CapDL spec generation is driven by a SDF to CapDL mapping, as expanded on in detail below. Most importantly, and crucially, the *verification of the SDF to CapDL mapping will guarantee correct initialisation*.

We note that writing the SDK in Python allows to reuse the well-tested and maintained, pre-existing Python CapDL bindings for this process. However, since it is not feasible to formally verify the functional correctness of the Python SDK directly, we perform instead a **Translation Validation**: the Implementation Relation correspondence between the *input SDF* and the *output CapDL* is shown post hoc, in each instance. (Implementation Relation is used here the same manner as earlier in Section 5.4.1.)

Recall that the static configuration of a Microkit-based system (i.e. the SDF spec used to generate it) is one of the constituent parts of the local state machine for the Microkit Section 5.4.2. The implementation relation relates the static configuration of the Microkit-based system to the capability distribution in the corresponding implementation kernel state. This means that exactly one capability distribution corresponds to a valid implementation of an SDF.

With this in mind, the next step is to transcribe the Implementation Relation suitably into Isabelle/HOL: The input SDF is imported into Isabelle/HOL as the static Microkit configuration, while the output CapDL is imported into Isabelle/HOL as the kernel state. A simple, automated proof script then verifies that the Implementation Relation is consistent between them.

6.2 Approach to formalising CapDL generation

As outlined above, the verification of the initialisation of the Microkit is based, firstly, on generating the CapDL spec from the SDF spec, and then on carrying out the Translation Validation. For the dedicated and formally/mathematically inclined reader, we set out in Chapter 7 the underlying formal definitions and a formalisation of a correct capability distribution implementation.

For the present, it is sufficient to realise that the specification defines data structures for storing the Microkit state and invariants. In particular, the Microkit Invariants correspond exactly to the Abstract Systems as defined formally in Chapter 7.

Moreover, the relation we built as part of our proof chain, `relation_cap_map` (which can be found in the download) captures the notion of accurately reflecting the SDF to CapDL

mapping. More explicitly, this relation `relation_cap_map` between the given Microkit Invariants and *Kernel Context* holds if the capabilities implied by the SDF describing the Microkit Invariants have been distributed correctly to the *Kernel Context* of the thread executing the current PD, as set forth in the SDF-capabilities mapping specification. In particular, this means in the formal language that this mapping relation faithfully manifests clauses 1–4 of Section 7.3.

We note that in the SMT-based (runtime) correctness proof we *assume* that every kernel call preserves this relation. In contrast, it is here with the Isabelle initialiser proofs, that we are obliged to *prove* that no call made by any user thread in a system that obeys the capability distribution of an SDF can put the system in a state that disobeys the capability distribution.

At present, the specifications are written in Haskell and then manually translated into Isabelle/HOL for the verification proof.

6.3 Choosing a CapDL loader

Above sections show that for user-system initialisation we can rely on a semantics-preserving mapping from SDF spec to CapDL spec. Hence, any of our CapDL loaders mentioned above in Section 6.1 can be chosen to initialise a Microkit-based system. However, at present they all come with their drawbacks and limitations:

- The original `capdl-loader`, while versatile, is not verified and does not embed the CapDL specification in a way that is compatible with the Microkit's SDK model. We note that the `capdl-loader` must be linked against a C version of the CapDL spec (which can be produced from the CapDL spec itself using an unverified Haskell tool).
- The new `rust-capdl-loader` is easy to use but also not verified (see Section 9.3 for detail).
- The formally verified `case-init` CapDL loader is presently too restrictive, in so far as it does not support the 64-bit Arm architecture nor the new MCS variant of the kernel (on which the Microkit is based). It is also cumbersome to use, as it requires the use of the kernel build system.

Chapter 7

Formal Framework for CapDL Generation

7.1 Preamble

In this chapter we delve into the detailed mathematical formalism underpinning the specifications and proof obligations when verifying the Microkit. While neither self-sufficient nor complete, it does provide insight into how particular components of the Microkit (which interfaces to the seL4 kernel) are captured with abstract objects in a specification language. Specifically, here we focus on the verification of our SDF to CapDL mapping (as introduced in opening comments of Section 6.1).

As described in Section 1.2.1 and Section 5.2, a Microkit-based system is composed of a few abstract, core objects and explicitly enumerated relationships between them. These core objects are protection domains (PD), communication channels (CC) and interrupts (IRQ), protected procedure calls (PPC), and memory regions (MR).

PDs can interact with each other expressed through various properties. For example, one such property is the presence of a channel, which serves as a means of authorising communications in the user's system specification. Interactions through channels can occur in two forms: sending notifications (`notified(c)`), which provide asynchronous signalling, and making protected procedure calls (PPCs, `protected(c)`), which allow for synchronous function calls between different PDs.

7.2 Abstract systems

In order to develop the mathematical framework to capture the Microkit's core objects with their characteristics, we begin by introducing the following definitions.

Page size k : $k \in \mathbb{N}^+$ (i.e. $k > 0$); assume k fixed for the purpose of the definitions below.

Valid protection domain identifier v : $v \in \{0, \dots, 62\}$;
 V denotes the set thereof.

Channel identifier c : $c \in \{0, \dots, 62\}$;
 C denotes the set thereof.

Endpoint (of a channel) (v, c) : $(v, c) \in V \times C$, where v is referred to as **end-PD**.

Channel $\{(v_1, c_1), (v_2, c_2)\}$: a channel is an unordered pair consisting of two **endpoints**

$(v_1, c_1) \in V \times C$ and $(v_2, c_2) \in V \times C$, with:

- different end-PDs, i.e. $v_1 \neq v_2$;¹
- at most one channel exists between two different PDs, i.e. given two channels $\{(v, c_1), (w, d_1)\}$ and $\{(v, c_2), (w, d_2)\}$ then $\{(v, c_1), (w, d_1)\} = \{(v, c_2), (w, d_2)\}$, which means $c_1 = c_2$ and $d_1 = d_2$;
- an endpoint of a channel cannot be attached to two different PDs, i.e. given two channels $\{(v, c), (v_1, c_1)\}$ and $\{(v, c), (v_2, c_2)\}$ then $(v_1, c_1) = (v_2, c_2)$.

Interrupts (IRQs) I : $I \subseteq \mathbb{N}$ and I finite.

Memory address a : $a \in k\mathbb{N}$ (i.e. $a \in \{0, k, 2 * k, 3 * k, \dots\}$);
 A denotes the set thereof.

Permission w : $w \subseteq \{W, X, C\}$, with W, X and R all distinct symbols;
 W denotes the set thereof.

Priority p : $p \in \{0, \dots, 254\}$;
 P denotes the set thereof.

Budget-period (x, y) : $(x, y) \in \mathbb{N}^+ \times \mathbb{N}^+$, where $x \leq y < 2^{64}$;
 B denotes the set thereof.

A mapped memory range, x comprises the following data:²

- a **base PD** b : $b \in V$;
- a **virtual address** a_v : $a_v \in A$;
- a **physical address** a_p : $a_p \in A$;
- a **permission** w : $w \in W$;
- a **size** n : $n \in k\mathbb{N}^+$, where $\forall m \in k\mathbb{N}$ with $m < n$: $a_v + m \in A$ and $a_p + m \in A$.

To visualise a mapped memory range including the role of size n , see Figure 7.1.

The key parameters are fixed as follows:
 page size $k = 4\text{Ki}$ (we drop “Ki” from now on)
 addresses $A = \{0, 4, 8, 12, 16, 20, 24, 28\}$
 virtual address $a_v = 4$
 physical address $a_p = 16$
 size $n = 8$

The requirement on the size n is:
 $m=0$: $a_v+m = 4+0 = 4 \in A$
 $a_p+m = 16+0 = 16 \in A$
 $m=4$: $a_v+m = 4+4 = 8 \in A$
 $a_p+m = 16+4 = 20 \in A$

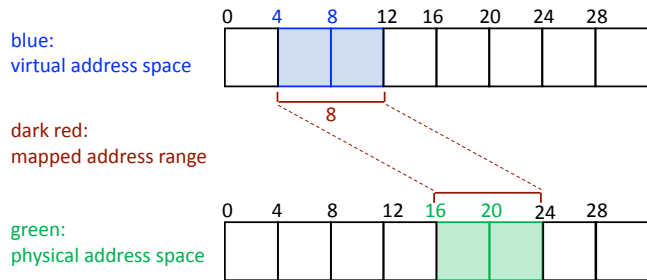


Figure 7.1: Example of a mapped memory range.

As per common usage we denote the field data for such structured objects using the *attribute* symbol “.” (e.g. the base PD b of the mapped memory range x is denoted by $x.b$).

An **abstract system** comprises the following data:

- a **set of identifiers of PDs with protected procedure call** V_p : $V_p \subseteq V$;

¹We note that self-channels are not required for the Microkit (in fact, they are not allowed), thus greatly simplifying concurrency handling down the track.

²For the demonstration purpose here we choose “range” over “region” purely for brevity and clarity of the abstract definitions; modifying for “region” will only require superficial adjustments.

- **set of channels** C_h (this is a subset of all unordered pairs of end-PDs, or loosely expressed $C_h \subseteq \{(V \times C), (V \times C)\}$);
- a **set of IRQ mappings** $I_m: I_m \subseteq I \times V \times C$ (i.e. an IRQ is mapped to a end-PD);
- a **finite set of mapped memory ranges** M ;
- a **priority map** $p_r: V \rightarrow P$;
- a **budget-period map** $b_p: V \rightarrow B$.

It is subject to the following conditions:

- All channels are disjoint, i.e. if $r, s \in C_h$, then either $r = s$ or $r \cap s = \emptyset$.
- IRQ ends and channel ends are disjoint, i.e. if $(i, e) \in I_m$ with $e = (v, c)$, and $r \in C_h$ with $r = \{(v_1, c_1), (v_2, c_2)\}$, then $e \notin r$, i.e. $(v, c) \neq (v_1, c_1)$ and $(v, c) \neq (v_2, c_2)$.
- Virtual addresses (of mapped memory ranges) are disjoint, i.e. for any two mapped memory ranges $x, y \in M$ with the same base PD $x.b = y.b$, then either $x.a_v < x.a_v + x.n \leq y.a_v < y.a_v + y.n$ or else $y.a_v < y.a_v + y.n \leq x.a_v < x.a_v + x.n$.

7.3 Accurate implementation by capability distribution

We are now prepared for formally capturing the notion that an seL4 machine state implements an abstract system. This in turn provides the formalism as alluded to in Section 6.2.

Take an abstract system as defined above in Section 7.2. Consider a finite set F of user ELF files and an assignment from PDs to these ELF files, $f: V \rightarrow F$.

We then define that (the capability distribution of) a kernel state **accurately implements** a given abstract system if the following conditions hold:

- For each PD $v \in V$, the file $f(v)$ is linked against the libmicrokit library and has that library's `main()` function as its entry point.
- There is an allocated thread control block (TCB) object in physical memory, referred to as the *Monitor TCB*, along with a synchronous IPC endpoint, referred to as the *Monitor Endpoint*, and a reply object, referred to as the *Monitor Reply*. The Monitor TCB is not suspended, and has priority 254 and max-priority 254. The Monitor TCB has a single CNode as its CSpace, which contains capabilities to Monitor Reply in slot 4 and to Monitor Endpoint in slot 74. The vSpace of the Monitor TCB consists only of the frames required to contain the Microkit monitor executable.
- For each IRQ $i \in I$ there is an IRQ handler capability that allows a thread possessing it to, firstly, set an endpoint which will be notified of the incoming interrupt i , and, secondly, to acknowledge received interrupts of the same number i .
- For each PD $v \in V$, there is a unique TCB object in physical memory, referred to as the v TCB, along with a unique endpoint for synchronous IPC, referred to as the v Endpoint, and a unique IPC buffer. The v TCB is not suspended, and has priority $p_r(v)$ and max-priority $p_r(v)$. There is a unique scheduling context, referred to as the v SC. The v SC has budget-period $b_p(v)$. The Fault endpoint of the v TCB is the Monitor Endpoint. There is a unique notification object, referred to as the v Notification Object.

If $v \in V_p$ then there is just the unique endpoint v Endpoint. The CSpace of the v TCB consists of one CNode, which contains the following capabilities:

- An unbadged RW capability to the v Notification Object in slot 1, referred to as the v Input Cap.
- An unbadged capability to the vSpace of the v TCB in slot 3.
- An unbadged capability to the v reply object in slot 4, referred to as the v Reply Cap.
- For each $u \in V$ and $c, d \in C$ with $\{(v, c), (u, d)\} \in C_h$: an RW capability to the w Notification Object in slot $10 + c$, badged with 2^d .
- For each $u \in V_p$ and $c, d \in C$ with $p_r(u) > p_r(v)$ and $\{(v, c), (u, d)\} \in C_h$: an RW capability to the u Notification Object in slot $74 + c$, badged with $2^{63} + d$.
- For each $i \in I$ and $c \in C$ with $(i, v, c) \in I_m$: an unbadged minted copy of the IRQ handler capability for i in slot $138 + c$.

The v VSpace contains mappings for the frames required to contain the user file $f(v)$. For each mapped memory range $m \in M$ with $m.b = v$, a frame is mapped with corresponding physical and virtual addresses, write permission if $w \in m.w$, executable permission if $x \in m.w$, and cached if $c \in m.w$.

DRAFT

Chapter 8

Our New Verification Tool Gordian

8.1 The process steps

The Gordian verifier is a newly developed, automated tool designed for the verification of functional correctness of the Microkit specifically and C programs more generally. Section 4.4 frames the context for which Gordian was developed with Figure 4.2 highlighting the central role it plays.

Before the Gordian verifier can be invoked, the target C source code must first be exported to the graph language GraphLang. This conversion is performed by the same C-to-Isabelle parser (from here on referred to as *C Parser*) used in the verification of the seL4 kernel itself. The C Parser reads the C input and emits code in Norbert Schirmer's SIMPL language [Schirmer, 2006], which, unlike C, comes equipped with a formal (big and small-step) operational semantics.

The SIMPL code is then converted to GraphLang using `SimplExport`, which is a verified tool originally implemented (and still used) as part of the translation validation of the seL4 kernel. The resulting GraphLang code is an unstructured graph program representing a control-flow graph, whose semantics refines the C semantics induced by the C Parser, and which consists of three sorts of nodes: *Basic* nodes which update local variables; *Cond* nodes which perform conditional jumps; and *Call* nodes which perform calls to other functions. See Figure 4.3 for an example of such a graph.

The Gordian verifier then processes its input in GraphLang along with a specification written in a combination of Python and SMT-LIB 2. Based on these inputs, Gordian generates a logical verification condition using a variation of Barnett and Leino's weakest precondition calculus for unstructured programs (see comment below in Section 8.2).

Finally, Gordian passes the verification condition to an SMT solver, which tries to verify its validity. The successful verification of the logical condition indicates that the C function under consideration conforms to its specification. If the verification fails, the SMT solver produces a counter-model.

We note that Gordian generates its verification condition in SMT-LIB 2, a common and standardised input and output language for SMT solvers. To a large extent, the logic theory used is the quantifier-free theory of arrays and bit-vectors, `QF_ABV`, the same, standard theory used for the binary verification proof of the seL4 kernel, and one which is supported in all contemporary SMT solvers. In some cases, we are resorting to the more flexible, but still

standard theories for quantifiers and algebraic data types. In practice, using SMT-LIB 2 allows us to increase both performance and certainty by running the verification condition through multiple SMT solvers.

8.2 The verification condition algorithm

We use a minor variant of the "weakest precondition for unstructured programs" algorithm of Barnett and Leino to determine the weakest conditions that need to be met in our control-flow graphs to prove correctness:

We first apply a loop-elimination transformation to the GraphLang graph, resulting in a control-flow graph that is free from cycles and that "traces correctness", whereby the latter means that this modified graph represents a program that is correct only if the original program was correct. (This means that if the modified graph is shown correct, we then know that the original graph must also be correct; however, if we obtain a negative result, we are not quite sure what it means for the original graph.)

The next step involves transforming the program into a control-flow graph in dynamic single-assignment form. This change makes it easier for SMT solvers to do their job. In *Gordian* we have implemented the loop elimination and the dynamic single-assignment transformation, along with a comprehensive test suite.

The final step applies the weakest precondition computation to the acyclic single-assignment control-flow graph to generate the verification condition to be handed to the SMT solvers.

DRAFT

Chapter 9

Existing Infrastructure Tools for Verifier Implementation

9.1 C Parser

In our toolchain, the same C Parser tool, `c-parser`, has been utilised that is used to create the Isabelle/HOL semantic model of the seL4 kernel. This tool translates the C source code into Schirmer's SIMPL programming language, which has a well-defined operational semantics in Isabelle/HOL. This, in turn, allows the same semantics to be used as that for the kernel verification. Additionally, SIMPL can be translated into the graph language GraphLang using the `SimplExport` tools, in a way that preserves the semantics.

The `c-parser` handles a reasonable subset of the C99 standard, and accounts for architecture-defined behaviours such as endianness, or the number of bits in an `int`. The most important limitations of the subset of C implemented by the tool are the following:

1. No `goto` statements
2. No fall-through cases in `switch` statements
3. No unions
4. No taking the address of local variables

These operations are already absent from the Microkit implementation, and adherence to these restrictions are not expected to become a constraint or difficulty in future development of the Microkit.

We note that an early version of the source code of the `handler_loop` in `libmicrokit` did contain one instance of the fourth limitation, where the address of the local variable `badge` had to be taken as a result of a peculiar design decision in `libseL4`. It turned out that a simple workaround was available by introducing an auxiliary global variable to store this `badge` (required changes: 4 LOC).

Due to the fact that the Microkit library and monitor are implemented in terms of other libraries such `libseL4`, which were not written with the `c-parser` in mind, we had to write preprocessing scripts before we could read the `libmicrokit` and `monitor` using the parser and successfully pump it through to graph language using `SimplExport`. Options to automate this process in a way that is resilient to Microkit implementation changes is to be explored at a later stage.

Statistics

Taking all includes together, exporting `libmicrokit` yields a graph consisting of 5627 nodes, divided into 320 functions. However, 302 of these come from `libseL4` and other includes (of which only 6 are relevant to the verification, and which have been already covered with the `libseL4` verification process), leaving a total of 18 functions (about 110 nodes) to be verified.

The main handler loop consists of 42 nodes.

The corresponding numbers are slightly larger for the `monitor`, which has 24 functions to be verified, with a total of 648 relevant nodes.

Exports of `libmicrokit` and `monitor` can be found in the following exports directory.

9.2 GraphLang and SimpleExport

The GraphLang language (formerly known as SydTV-GL, Sydney Translation Validation Graph Language), used to describe control-flow graphs was originally developed for, and forms an integral part of, the seL4 kernel's translation validation and worst-case execution time analysis proofs.

Our Gordian verifier also uses GraphLang as its foundation. As per Figure 4.2 and Section 4.4.1, in one of our verification process steps we employ the `SimpleExport` tool to obtain the intermediate artefact written in this graph language. This approach allows us to create an implementation that is compatible with previous advancements and to reuse existing code for parsing GraphLang. We improved the type-safety of the existing GraphLang toolchain by adding new interfaces with additional checks and constraints. These fine-tunings not only help to prevent programming errors in the new developments, but can be upstreamed to increase the overall trustworthiness of our toolchain.

The external translation validation tools are undergoing development as part of the *Binary Correctness and Multicore Verification* topic area. Once concluded and stable, we expect resulting improvements to be suitable to be incorporated into the translation validation version of our toolchain Gordian.

9.3 Rust CapDL loader

Nick Spinale has recently developed a new CapDL loader implemented in Rust [Spinale, 2023], accompanied by a simple, self-contained tool that serialises a CapDL spec and adds it to the pre-compiled loader binary. This scheme aligns it well with the Microkit ecosystem. This Rust CapDL loader also supports a configuration whose operation is more akin to that of the unverified C loader today, allowing for a simpler and space-efficient loader, as there is no deserialisation at runtime. In fact, under this configuration, the loader does not even require a heap allocator. For our purpose, though, it is noteworthy that, to date, this Rust CapDL loader is not as yet verified.

Importantly and in summary, our verified version of the Microkit does, along with the verified CASE initialiser, support (and is integrated with) Spinale's new Rust CapDL loader.

Chapter 10

Worked Examples Walking Through The Verification Steps

DRAFT

To be done

Chapter 11

Limitations

11.1 Microkit versions

The verification of the Microkit is based on an implementation version of the Microkit as was available at the time of beginning the project mid 2022 and therefore differs in some key details from the current upstream Microkit version available in the public git repository as of mid 2023.

Kernel limitations: Non-MCS

The upstream Microkit is built on top of the mixed-criticality scheduling (MCS) seL4 kernel. However, the MCS kernel does not yet have a full verification story. In particular, the verified `case-init` loader neither supports MCS no 64-bit architectures.

In order to obtain a complete verification story, the verification supports not only the upstream Microkit, but also a fork that uses the non-MCS 32-bit kernel. This fork will be redundant once MCS verification and `case-init` have caught up.

Architecture limitations: 32-bit

Similar caveats hold for the architecture: The verified system initialiser does not presently support 64-bit architectures but only 32-bit architectures.

SDF to CapDL mapping: platform-aware

Ultimately, the functionality for automatically generating CapDL output based on the user-provided system SDF spec should target all platforms supported by the Microkit.

However, the Microkit code is by nature platform-aware (as opposed to platform-agnostic), and the Microkit environment currently supports only a few development boards (essentially a limited set of AArch64 boards). The specification may have to be generalised to be more platform-agnostic as support for new development boards expands.

Build system

A motivator for using Spinale's new Rust CapDL loader is that it had been designed with the SDK model in mind. The existing CapDL loader written in C required re-compilation every

time a system designer would change the capability distribution of their system. In addition, the C CapDL loader was tightly integrated with the rest of the seL4 build system, which meant that trying to fit it in an SDK model posed significant friction. Spinale's new Rust CapDL loader do not exhibit these down-sides.

11.2 Gap in end-to-end proof

11.2.1 Verifying MSpec

The key assurance gap for end-to-end proof is **Proof ③** in Figure 1.4: We still need to demonstrate that MSpec, the kernel specification relevant to the Microkit, is a *provably correct abstraction* of the kernel's abstract specification, ASpec, when we manually derive MSpec from ASpec to make it suitable for SMT solvers.

11.2.2 Haskell to SMT-LIB 2 transcription

In our proof pipeline of the Microkit verification process (Figure 4.2) we note that the transcription of the spec in Haskell to SMT-LIB 2 is a *manual* process. More specifically, it is a 2-step process of first manually transcribing the Haskell spec into Python, and then translating by Gordian to SMT-LIB 2.

At the time, the use of Haskell for the formal specification allowed for efficient initial prototyping and development. Meanwhile, the eventual implementation of the Microkit verifier is SMT-solver based (essentially to prove functional correctness), while Isabelle/HOL is to be used for proofs of properties relating the MSpec with the ASpec. Accordingly, a manual translation of the Haskell specification into Isabelle/HOL is required with, eventually, an *automated* translation from the Isabelle/HOL specification to SMT-LIB 2 that is simple enough to demonstrate the correspondence between these two versions of the specification.

However, once the specs in Haskell have been translated into SMT-LIB 2 and Isabelle/HOL, as required, these Haskell specs are no longer crucial to the verification process. We will therefore no longer maintain the Haskell spec, and consider the Isabelle/HOL and SMT-LIB 2 specs authoritative, with a simple *automated* translation directly between the two serving to assure their correspondence.

11.3 Threats to validity

There are a number of assumptions on which the proofs reported here are based. Specifically these are:

- We prove *functional correctness* only between the induced semantics for the C code and its specification, and so the *compiler and linker need to be trusted*. This gap can be bridged by combining our Microkit tool with the existing seL4 binary-verification toolchain. This toolchain proves that the seL4 binary has the same semantics as the verified C code, and thus ensures that the seL4 kernel proofs apply to the kernel binary. In the same way, that toolchain should be able to extend the verification of `libmicrokit` to the binary code.

- The proof work is done by SMT solvers. We therefore *assume that the SMT solver used is functionally correct* and is invoked correctly.
- Finally, one also has to make *some bottom-level assumptions about the physical world and other code* running in the system. Tackling these have to be left to future work (where possible) or have to be validated by empirical means. If these assumptions are not met, faults can still occur.

In our case, the assumptions are that the *hardware* works as specified by the manufacturer, the *kernel has been loaded* correctly, and that the *libraries* outside the scope of the verification project, such as `libseL4`, also satisfy the properties stated in the spec. Furthermore, the correct initialisation by `system_initialiser` is assumed unless the verified `case-init` loader is used.

DRAFT

Chapter 12

Achievements and Impact of Verification

12.1 Achievements: What we have proved

The automated verification of the Microkit implementation proceeds via multiple stages (see Figure 12.1) and reuses a number of tools and libraries that were developed at the time for the kernel verification effort.

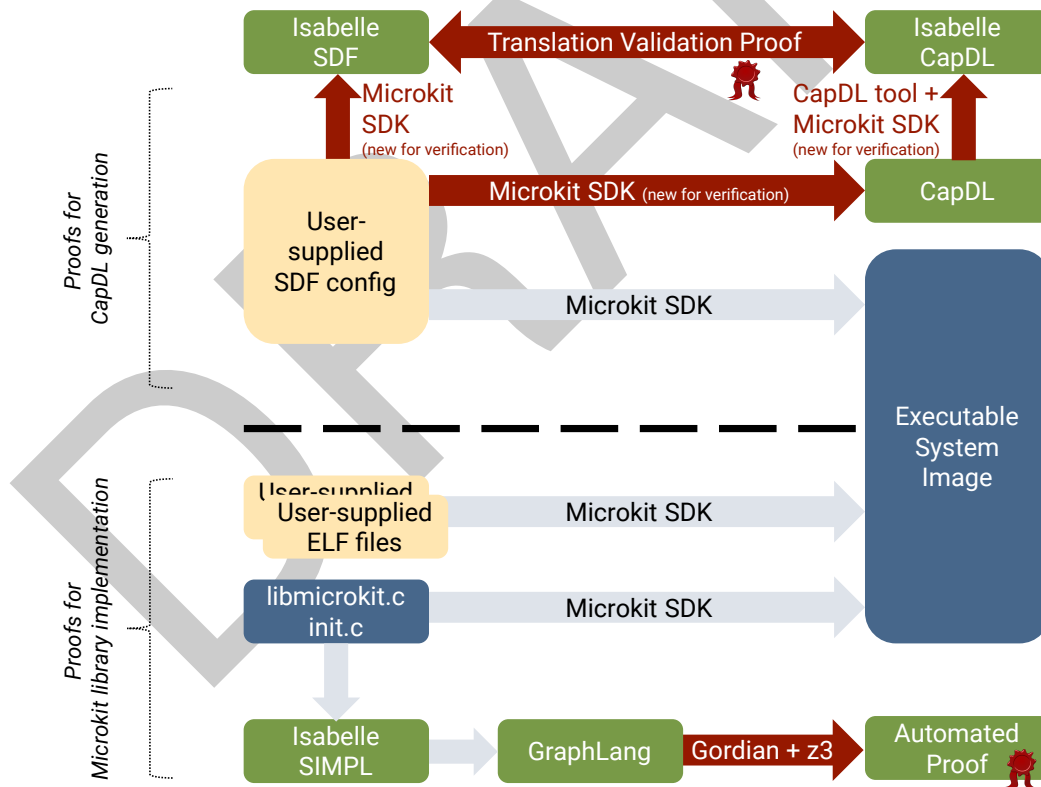


Figure 12.1: Summary of the verification processes for the translation validation of the generated CapDL export and for the automated verification of the Microkit library. In yellow are user-supplied files, in green formal artefacts created as part of the Microkit toolchain and in blue the informal ones. Red arrows indicate the new translation and verification tools plus expanded Microkit SDK developed over the course of this verification project, while light grey arrows are the result of prior work. The dashed line separates the two sets of proofs.

Functional correctness We have verified a Microkit implementation, encompassing both the library (`libmicrokit`) and monitor task (`monitor`) components, using *push-button methods*: Upon successful completion, the Gordian verifier confirms that a provided C function is functionally correct according to its specification.

Absence of Undefined Behaviour The proof chain with Gordian ensures the absence of common programming errors, including:

- No null pointer dereferences
- No incorrect use of dynamic memory during program execution (e.g., no ill-typed or dangling pointers, no out-of-bounds errors)
- No arithmetic overflows and exceptions (e.g., no signed integer overflows, no division-by-zero, no invalid bit shifts, no invalid conversions)
- No other undefined behavior (e.g., not trying to use values of uninitialised local variables)

Non-termination allowed It is important to note that in our specifications for the Microkit components, non-termination is always an allowed behavior.

- For example, the handler loop is specified to never terminate, and the verifier explicitly confirms this property.

Re-use of Gordian The design and development of our automated verification tool for the task, Gordian, has been built with the intention to be re-used in the verification of future projects built within the Microkit framework.

12.2 Impact

Error elimination

While the final verification succeeded without finding new bugs or errors in the implementation, we identified and eliminated two errors during the specification and verifier development process, improving the overall quality of the code.

As a result, the insights gained during the formal specification and later verification process have already raised the assurance levels of the Microkit and the surrounding ecosystem.

Rust CapDL loader

Moreover, the desire for better assurance led Nick Spinale to develop a new CapDL loader written in Rust [Spinale, 2023]. Since Rust ensures memory safety at compile time using its ownership mechanism and built-in borrow checker, the Rust CapDL loader serves as a safer *drop-in* replacement for the original C CapDL loader – be this for Microkit-based systems or, in fact, for other CapDL-based systems, legacy or otherwise.

Access Control Policy

The proofs of the CASE initialiser *go higher* than merely showing that the initialiser correctly sets up the capability distribution of the system according to a given CapDL, namely in the following sense: An Access Control Policy (“ACP”) can be derived from a CAMkES system

specification; furthermore, an ACP can also be derived for the initialised system. The CASE initialiser proofs are then able to establish that the ACP of the system specification is satisfied by the initialised system.

We are currently investigating whether we could provide a similar policy preservation proof for Microkit-based systems.

While the current executable implementation of the SDF-CapDL translator is not formally verified (and not easily formally verifiable), an ACP refinement proof sidestep this issue: We will not have to worry about our translator being 'correct' in some lower-level sense, as long as in each particular instance we can prove that the ACP defined in the SDF config file is in fact the same as the one defined by the CapDL that will be used to initialise the system.

If, however, the current investigation into the ACP proofs reveals that showing the analogous property for the SDF would be too much effort for the remaining time in this Phase, we can alternatively use the formalised SDF semantics to develop a verifiable CapDL generator directly in ML.

DRAFT

Bibliography

- Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Lisbon, PT, September 2005. 18
- Aditi Barthwal and Michael Norrish. Verified, executable parsing. In *European Symposium on Programming*, pages 160–174, York, March 2009. Springer. 17
- Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, September 2003. 4
- Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, Hungary, March 2008. Springer. 18
- Gernot Heiser, Lucy Parker, Peter Chubb, Ivan Velickovic, and Ben Leslie. Can we put the "S" into IoT? In *IEEE World Forum on Internet of Things*, Yokohama, JP, November 2022. 3
- C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21: 666–77, 1978. 4, 6
- Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997. 4
- Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014. 3, 14, 19, 22
- Ihor Kuz, Gerwin Klein, Corey Lewis, and Adam Christopher Walker. capDL: A language for describing capability-based systems. In *Asia-Pacific Workshop on Systems (APSys)*, pages 31–35, New Delhi, India, August 2010. 5, 32
- Jochen Liedtke. On μ -kernel construction. In *ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995. ACM. 3
- Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities: A principled, light-weight OS mechanism for managing time. In *EuroSys Conference*, Porto, Portugal, April 2018. ACM. 11
- Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and secure inter-process communication for microkernels. In *EuroSys Conference*, Dresden, DE, March 2019. ACM. 3
- Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina

- Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 252–269. ACM, 2017. 4
- Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *ACM Symposium on Operating Systems Principles*, pages 225–242, October 2019. 4
- John Ousterhout. Why threads are a bad idea (for most purposes). USENIX Technical Conference (Invited Talk), January 1996. 4
- Lucy Parker. High-performance networking on seL4. BSc(Hons) thesis, School of Computer Science and Engineering, Sydney, Australia, November 2023. 3
- Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a microkernel, a big lock is fine. In *Asia-Pacific Workshop on Systems (APSys)*, Tokyo, JP, July 2015. ACM. 4
- Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006. 17, 39
- Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 471–481, Seattle, Washington, USA, June 2013. ACM. 4, 18
- Thomas Sewell, Felix Kam, and Gernot Heiser. High-assurance timing analysis for a high-assurance real-time OS. *Real-Time Systems*, 53:812–853, September 2017. 15
- Nick Spinale. Rust-seL4 CapDL loader, 2023. URL <https://github.com/seL4/rust-seL4/tree/main/crates/seL4-capdl-initializer>. 33, 42, 48
- Yong Kiam Tan, Magnus Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *International Conference on Functional Programming*, page 14, Nara, Japan, September 2016. 5, 33
- Wikipedia. Executable and linkable format, 2001. URL https://en.wikipedia.org/wiki/Executable_and_Linkable_Format. 7