Formal Verification of a Component Platform

Matthew Fernandez

A thesis in fulfilment of the requirements for the degree of

Doctor of Philosophy

UNSW
AUSTRALIA

School of Computer Science and Engineering

Faculty of Engineering

March, 2016

Abstract

The function of software used to be calculation; mechanising what was previously done by hand. Now it runs our communication networks, mass transportation and medical support. Yet we still build large software systems as if they were small, easily comprehensible tools.

The right to manage our safety and security should not be handed over lightly. When a program has the ability to compromise our security or injure us, we should demand evidence of its correctness. Formal software verification has demonstrated how to reliably and repeatedly build safe and secure high-assurance systems, to a standard not achievable using other techniques. Yet it remains underused due to perceptions that it is expensive and time intensive to apply.

In this thesis we demonstrate how to scale formal software verification beyond its current limits using component-based software engineering. By leveraging the strong isolation boundaries made possible by the CAmkES component platform operating on the seL4 microkernel, we decompose system verification along lines that correspond to the system architecture. The parallels between proof obligations and system architecture aid the designer's intuition and allow easing verification challenges through architectural refactoring.

To uphold the engineering abstraction that a component platform provides, we demonstrate a fully automated process for verifying functional correctness of platform-generated code and the correct initialisation of a CAmkES system. The system designer no longer needs to trust that the platform's mechanisms do what they claim, as is the case with other existing component platforms. We also fully automate the production of an access control policy for a component-based system, allowing the designer to move seamlessly from architecture layout to security analysis, with knowledge that our verification guarantees a faithful translation.

The techniques in this thesis represent novel contributions to the fields of component-based software and formal verification. To our knowledge, they provide the most trustworthy development environment for high-assurance, componentised software systems in existence today. By extending the power and scope of formal verification, we lower the cost of its application and enable the development of safer and more secure software.

Acknowledgements

There is a persistent myth that a PhD thesis is the work of a single individual. The reality involves a cast of family, friends and advisors who act as silent co-authors. When a student submits their research proposal, these other parties also begin a PhD, despite never signing their consent to the proposal. This preface serves as a belated attempt to thank those who unwittingly accompanied me on this journey.

Throughout my PhD, I have been privileged to be co-supervised by outstanding researchers, in the form of **Gerwin**, **June** and **Kevin**. Their abilities to conceptualise and pursue both short term and long term research goals are, in my experience, unparalleled.

I could not have asked for a more capable primary supervisor than **Ihor**. Many of the research skills I have today, I learned from working with him. I am deeply indebted to him for the example he has shown and the wisdom he has passed on. He has never failed to have answers to my questions and questions to my answers.

I have shared many highs and lows with the **NICTA Trustworthy Systems Team** during my PhD. There are too many people to thank by name, but they collectively made my PhD an unforgettable experience. It is safe to say I learnt more from them about systems engineering, formal methods and coffee than I ever have in a classroom.

Finally, I would like to thank my family, **Anne**, **Cyrus**, **Gabe**, **Joe**, **Melanie** and **Thea**. They are a constant source of inspiration and encouragement to me. I am regularly humbled by their patience and intelligence. I strive to be more like them each day.

Contents

Chapter 1

Introduction

Software has ever-increasing prominence in our lives and power over them. Where programs used to be solely tools for scientific calculation, they are now our instruments of communication, transportation and medical support. With this elevated role comes increased responsibility. The consequences of software failure used to be inconvenience and lost productivity. Now they include privacy compromises, injury and even death.

Over the course of 2009 to 2011, Toyota issued several recalls of their cars in relation to brake defects (CBS News 2010). A 2013 court investigation found these brake defects, that led to a number of deaths, could be attributed to errors in the design of Toyota's software (Parrish 2013). This design was the result of well-established, but clearly insufficient software engineering processes. Sadly this example is not atypical of modern safety-critical software systems.

When the cost of failure is so high, we should be applying every tool we have for guaranteeing software correctness. While this includes validation techniques such as unit and integration testing, these alone can only ever prove the presence of bugs, not their absence. A different family of techniques known as formal software verification provides the complementary approach to testing. Rather than searching for bugs in a program, formal verification attempts to prove the *correctness* of the program, and by extension the absence of bugs.

The benefits of software verification techniques are recognised widely enough to be required by some official safety and security standards. The leading recognised safety standard for automotive, MISRA, requires the use of formal methods at its highest tier.[1] The leading safety standard for aviation, DO-178C, allows formal methods to substitute for testing. Similarly, the most prominent set of security standards, the Common Criteria, requires formal verification at its highest level of certification, Evaluation Assurance Level 7 (Common Criteria Recognition Arrangement 2012). Nevertheless, software verification remains underused in high-assurance environments. The primary motivation for this neglect of software verification is its perceived cost and impracticality. This has been shown previously to be a misconception in relation to the typical cost of certified software (Klein et al. 2014), but this misperceived cost is often enough to discourage its use.

It is our position that this status quo is untenable. If we knew of a technique for building safer bridges, it would be soon incorporated into building codes and made mandatory for new construction. Life-saving technologies like anti-lock braking have regularly been made compulsory soon after their invention. Software is now a

[1] It is relevant to note that, despite these standards existing, very few legal jurisdictions require adherence to any level of the MISRA standard for production automobiles, let alone its Safety Integrity Level 4 that mandates formal verification. For example, the United States' vehicle regulation standard (National Highway Traffic and Safety Administration 2016) does not require any level of MISRA compliance. Further guidance on the use of formal verification in the automotive domain is provided in ISO 26262 (*ISO26262: Road Vehicles – Functional Safety* 2011) and its precursor, IEC 61508 (*IEC 61508: Functional Safety of Electrical, Electronic and Programmable Electronic Safety Related Systems* 1998). However their terms are imprecise and they are inconsistently applied (Sexton 2013). In contrast, it is much more common for legal jurisdictions to require commercial aviation companies to comply with DO-178C or its predecessors (Transportation Research Board 2012).

factor in the safety and security of high-assurance systems, as influential as mechanical components. It is time that high-assurance software be held to the same standards we demand of the physical parts that make up a safety- or security-critical system.

In this thesis, we propose a divide-and-conquer technique for reducing the verification challenge in large systems, and thus decreasing the associated cost. By leveraging the principles of Component-Based Software Engineering (CBSE) (Crnkovic 2001; Szyperski 1997), we decompose verification along the same lines as the engineering abstractions of the system architecture. We provide strong, provable isolation boundaries between components in a system, allowing the system architecture to influence and reduce the verification effort. The proof obligations in such a system are made more intuitive to the system designer by mirroring the componentisation of the architecture and all obligations related to generated platform code are discharged automatically.

1.1 Approach

In our work we build on two existing pieces of software, the formally verified seL4 microkernel (Klein et al. 2009) and the CAmkES component platform (Kuz et al. 2007), that are intended as the basis for high-assurance, embedded software systems. We use these as the foundation of an environment for the construction of verified safe and secure software. While the implementations we present use mechanisms specific to seL4 and CAmkES, our underlying techniques are applicable more broadly. Our results could be repeated on another capability-based operating system and static component platform, though we know of no others currently existing that are as well suited to our aims and provide equivalent foundational guarantees.

We aim to ease two orthogonal tasks in the construction of component-based systems: trust in the system of components described by the designer and trust in the component platform itself. We target the C programming language, which avoids the need to further trust a language runtime. However, this induces a different set of challenges. As depicted in Figure 1.1, code generated by the component platform is compiled with user-provided code. The lack of memory safety in C means that the user-provided code and generated code can affect each other and errors in either are not constrained in scope by the language. This leads to additional complications when verifying such code.

To address the verification of components themselves, we provide a series of distinct, gradated techniques for formally verifying the correctness of a system of components. The corresponding formalisms are not connected to each other, but rather provide models of the same component-based system with increasing degrees of expressivity, fidelity and assurance. This increasing level of detail comes at the cost of decreased automation. Each has its own utility and the combination allows the system designer to tailor their choice of formalism to their desired class of properties or level of assurance.

The first of these techniques, described in Chapter 4, defines a type system for component system architectures and a process for automatically generating a representation of a given component system in this type system. Properties that are true of the system architecture can be proven fully automatically in this formalism. This work has been the subject of a previous paper (Fernandez, Kuz, et al. 2013) and technical report (Fernandez, Klein, et al. 2013).

To enable proofs that rely on the runtime behaviour of components, in Chapter 5 we provide a more complex formalism capable of representing the execution of user-provided code. This formalism still provides significant automation, only requiring the user to specify the trusted component instances in their system. Proofs within this formalism generally require some manual steps, but can show more complex properties of a system. This work has been the subject of a previous technical report (Fernandez, Gammie, et al. 2013).

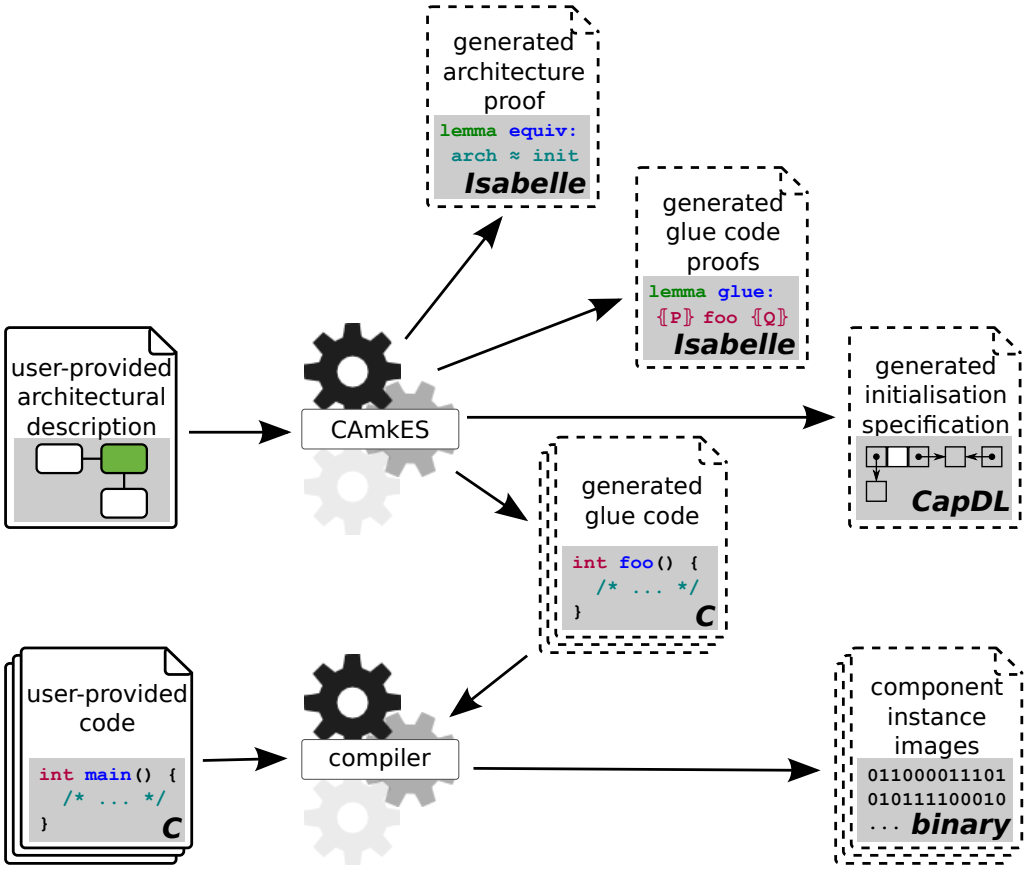

Figure 1.1: Component platform work flow

Finally, Chapter 6 provides a formalism that represents the behaviour of components by a direct translation of their source code. This formalism is strongly guaranteed to be faithful to the implementation's behaviour, at the expense of greater reasoning effort to discharge correctness goals. This work has been the subject of a previous paper (Fernandez, Andronick, et al. 2015a) and technical report (Fernandez, Andronick, et al. 2015b).

All three of these formalisms led to the discovery of component platform bugs that have since been fixed. In particular, the last formalism, and indeed any pervasive component system verification effort, requires deep justification of the correctness of the component platform itself. To remove this obligation from the system designer, we demonstrate a technique for the automatic generation of code-level correctness proofs of generated code (depicted as "generated glue code proofs" in Figure 1.1). To guarantee that this code is operating in the environment intended, we also demonstrate an automated proof of correct architecture and initialisation of a component platform in Chapter 7 (depicted as "generated architecture proof" in Figure 1.1). These proofs, which integrate seamlessly with manual proofs of the components themselves, reduce the challenge of component system verification to simply demonstrating the correctness of the hand written code provided by the system designer themselves.

Our approach enables a machine-checked, code-level proof of the correctness of an entire software system, providing the highest known level of assurance currently achievable in safety- and security-critical software. The composed guarantees of seL4 and CAmkES raise the bar for the scale of formal software verification. This in turn makes possible the next generation of larger high-assurance software at a reduced cost.

1.2 Contribution

This thesis contains four distinct pieces of work, each of which has a chapter devoted to its exposition:

- A static architecture model of component-based systems (Chapter 4);
- A dynamic behavioural model of component-based systems (Chapter 5);
- A proof-producing process guaranteeing the correctness of platform-generated code (Chapter 6); and
- A proof of the correct architecture implementation of component-based systems (Chapter 7).

Taken together, these represent four novel contributions to the field of computer science, detailed in the following four sections.

1.2.1 Component Platform Correctness

Component platforms provide a development environment and introduce generated code that cooperates with hand written, user-provided code to achieve the system's goals. The collaborative control flow between hand written code and platform mechanisms requires that both are correct for the system to operate correctly. Whether the user realises it or not, their system implicitly depends on code that the platform has introduced on their behalf.

There have been many attempts at providing correctness guarantees for component platforms, but previous work operates on platform *models*, leaving a significant semantic gap between abstract specification and implementation. To our knowledge, our work is the first result that guarantees code-level correctness for the platform mechanisms. In contrast to proofs on a model of the component platform, we demonstrate the highest level of assurance in a component platform currently available.

1.2.2 Automated Verification of Generated Code

Gaining confidence in the correctness of a software system that mixes hand written and generated code requires reasoning about the generated code – artefacts with which the system designer is unfamiliar. A code-level proof of correctness often necessitates an intimate familiarity with the code in order to have an intuition for why it is correct. Lacking this intuition, such reasoning can be challenging if not impossible for system designers.

To resolve this imbalance, we provide automatically generated proofs for generated code. This is possible by leveraging properties of its regularity. Now the designer is not only freed from this unfamiliar reasoning, but also no longer needs to trust the correctness of the code generator. This symmetry with the system architecture naturally leads to points of interconnection between hand written and generated proofs that are more familiar to the system designer. We effectively remove any assumptions on the correctness of the component platform by automatically demonstrating it with formal artefacts. To our knowledge, no other existing component platform provides this utility.

1.2.3 A Methodology for Scaling Software Verification

While formal software verification is a well-established technique, its application in the form of code-level proofs of realistic systems is still relatively immature. Existing verification techniques are extremely limited in the size of the code bases they can be applied to. This has impeded the uptake of formal verification in the

domains of safety- and security-critical software where it is most needed, as existing code bases are typically orders of magnitude larger than what is feasible to verify.

The techniques we demonstrate allow partitioning and de-duplication of the verification challenge. The system architecture boundaries can be leveraged as natural cut points in proofs and verified components can be instantiated multiple times without repeating proof efforts. Not only does this make individual verification obligations more tractable, it removes some entirely, by formally defining a bound on the effect one partition can have on another. Rather than attempting to scale formal verification by optimising individual proof scenarios, we aim to have the system designer verify less. Using the system's natural decomposition and by providing increased automation, we enable software verification on scales not previously achievable.

1.2.4 A Path to Whole System Assurance

While the correctness of general purpose platform mechanisms is a necessary condition for building high-assurance systems, the ultimate property of interest to the designer is the correctness of their system as a whole. Traditionally this has required a monolithic, time-intensive effort to validate and verify an entire system. By building on the results of the seL4 microkernel verification and verified system initialisation (Boyton 2014), and by producing reusable formal artefacts, we show how to extend our work to cover full functional correctness of industrial scale systems.

If we further consider non-functional correctness properties, the majority are predicated on functional correctness itself. For example, an analysis of the worst case execution time of a system is worth relatively little without an accompanying guarantee of functional correctness. By providing a feasible strategy for showing functional correctness for an entire system, we enable the extension of high-assurance requirements to other aspects of a program. We see our current work as a step on the path to more comprehensive whole system assurance.

1.3 Summary

The ability of software defects to compromise our safety and security is ever-increasing, while the maturity of software development techniques and their adoption have not kept pace. As the role and responsibility of software grows, we must strive to improve our approach for building high-assurance systems. Formal software verification offers the greatest promise for the construction of robust programs, yet its scalability has proven challenging to its adoption by many branches of industrial software engineering.

This thesis describes new techniques for scaling formal verification beyond its current limitations. We demonstrate a comprehensive approach for the verification of a component platform, enabling both greater trust in a general purpose embedded environment as well as the decomposition of a system verification effort. By taking an approach of "generated proofs for generated code," we reduce a system designer's task to verifying their own hand written code, and provide them with a proof environment that automates key parts of their reasoning. The outcomes of our work provide tools for building safer and more secure software, on a scale not previously possible.

Chapter 2

Background

This chapter provides an overview of the technologies on which the work in this thesis is based. This essentially falls into three distinct areas: microkernels, component-based software engineering and interactive theorem proving. We also discuss and disambiguate the term "correctness." The correctness of a component platform is a vague concept and it is important to enter into the following chapters with a more precise understanding of what it is we are attempting to achieve.

Note that the tools and technology discussed in this chapter are predominantly *not* the work of the current author, but are simply provided for context. The exception is the CAmkES component platform, whose current implementation was primarily the work of the author. However, the implementation is not considered a primary contribution of this thesis.

2.1 Microkernels

At the base of a software system is an operating system, a general purpose program that is responsible for initialising the hardware platform and providing general runtime services for other programs. The core code of an operating system is the kernel, a piece of software that runs in the privileged mode of the processor. For the high-assurance, embedded environments we are targeting, a desktop operating system with a monolithic kernel is unsuitable. Any incorrect or malicious code in the operating system kernel has the ability to compromise the entire system, and thus monolithic kernels have a large attack surface. For this reason, it is desirable to minimise the amount of kernel code by applying the Principle of Least Authority (POLA) (Saltzer 1974).

The idea of a microkernel takes POLA to the extreme by evicting everything from the kernel except functionality that is impossible to implement efficiently outside it (Brinch Hansen 1970). This approach minimises the trusted computing base of a system, the set of code which must be correct for the system to operate as intended. A microkernel serves as an ideal platform for high-assurance systems because it is able to leverage the hardware isolation mechanisms of virtual memory and provide a minimal attack surface to hosted processes. Adoption of microkernels in their early history proved slow due to perceived inherent performance limitations, but these were shown to be artefacts of the specific microkernel systems of the time and not a fundamental weakness of the idea (Liedtke 1993). Nowadays, microkernels are used as the basis for several commercially deployed high-assurance systems (Hieb and Graham 2009; Laroux and Graham 2009; "Microvisor Products - General Dynamics Mission Systems" 2016; National Security Agency and Center 2010).

2.1.1 Capability-based Access Control

As a general purpose environment, an operating system kernel typically hosts multiple, potentially uncooperative programs. In such an environment, it is necessary to have a mechanism for managing access to shared resources. Desktop operating systems generally have only coarse-grained mechanisms for granting access to resources and rely on interactive intervention from a user. Most implement a security model that can be categorised as discretionary access control, wherein authority is easily passed between parties and it is up to the user to constrain permissions. They generally assume that failures due to resource exhaustion are acceptable. Such a model is unsuitable for embedded, high-assurance environments where there may be no user to intervene and even transient failures can be catastrophic.

An alternative technique for much finer-grained access control is based on the idea of "capabilities." A capability is an unforgeable token granting access to a specific resource (Dennis and Van Horn 1966; Levy 1984). An operating system built on this security model requires programs to have capabilities to any shared resources they use. Discretionary access control can be implemented with capabilities, but it is also possible to use capabilities to implement mandatory access control, wherein authority is defined by a set of rules enforced by the operating system. Capabilities, as an abstraction, are powerful in their ability to support formal reasoning while maintaining a strong connection to a system's implementation (Shapiro 2003):

> *The capability model is currently the only model offering a semantics that allows us to reason all the way from user-level object interactions down to machine-level instructions in a uniform and consistent way.*

Numerous capability-based operating systems have been proposed or implemented previously (Feiertag and Neumann 1979; Hardy 1985; Shapiro 1997; Wulf et al. 1974). There has traditionally been some skepticism directed towards capability-based systems, primarily motivated by their perceived inflexibility and inferiority to access control lists (Boebert 1984; Chander, Dean, and Mitchell 2001). More recently this has been refuted by a critical mass of counter arguments and implementations providing counterexamples to these perceptions (Miller, Yee, and Shapiro 2003; Shapiro 1999; Shapiro, Smith, and Farber 1999).

2.1.2 seL4

While the microkernel approach decreases the size of the trusted computing base, it does not inherently increase the safety and security of the remaining kernel code. The seL4 microkernel was designed to meet this challenge by providing an L4-based (Liedtke 1995) implementation that has been proven to be functionally correct (Klein et al. 2009). This proof guarantees that all system calls terminate, the kernel never crashes or deviates from its specification, and it never executes any code that has undefined behaviour; for example, dereferencing a null pointer (Klein et al. 2014). This prevents a large class of the most prevalent bugs in embedded systems.

The seL4 verification was later extended to prove that the microkernel preserves the integrity of hosted processes (Sewell et al. 2011). Furthermore, the design of seL4 enables it to be configured as a separation kernel (Alves-Foss et al. 2006; Rushby 1981; Rushby 1984), partitioning a system into strongly-isolated domains. This ability to provide proven-correct isolation boundaries is well aligned with the aims of a component platform – component separation and system decomposition – making seL4 the ideal base for a high-assurance, component-based system. We are not aware of any other general purpose operating system kernel that provides as strong safety and security properties as seL4.

At present, seL4 supports two hardware architectures: ARM and x86. The verification work in the previous paragraphs applies specifically to seL4 targeting an ARMv6 platform. Following this lead, our work is also specific to ARMv6, though this is primarily to maximise its utility. Should the seL4 verification be extended to another hardware platform in future, it would be a trivial matter to adapt our work to the new platform.[1]

All kernel-managed resources in seL4, including memory, are represented by explicit "kernel objects." All memory available after the kernel has booted is provided to the first userspace task (the "root" task) as untyped memory when the kernel transfers control to the task. From there, it is the responsibility of the (system-specific) root task to split and retype the untyped memory into other objects and capabilities to construct a system. The kernel object types are:

- address space identifier (ASID) pool – Each virtual address space requires an assigned identifier. These identifiers are produced from an ASID pool.
- asynchronous endpoint[2] – A form of non-blocking data-less interprocess communication (IPC) can be performed over an asynchronous endpoint. This communication is similar to the traditional concept of signalling, or can be seen as equivalent to a set of binary semaphores.
- capability node (CNode) – Capabilities are stored in capability nodes. CNodes are power-of-two sized from 2^2 to 2^{27} slots. The contents of CNode slots and their addressing is expanded on below.
- frame – Creation of a virtual address space requires that some untyped memory be retyped into frames, units of physical memory, and then mapped into a page directory or page table. The possible frame sizes are hardware dependent. On ARM, these are 4KB, 64KB, 1MB and 16MB.
- interrupt handler – Each device interrupt can be associated with an interrupt handler, that in turn is bound to an asynchronous endpoint. Following this, interrupts from the device are delivered by seL4 to the endpoint and a userspace device driver can receive and process them by waiting on the endpoint.
- interrupt control – A single interrupt control object exists that allows creation of interrupt handlers. It is not possible to copy or create more interrupt control objects because it allows privileged operations. The root task is expected to setup all interrupt handlers or delegate the interrupt control object to a trusted second stage loader.
- page directory – The top level paging structure of the hardware is modelled directly as a kernel object and referred to as a page directory. Page directories are containers for page tables and/or frames.
- page table – Page tables are containers for frames, that must be mapped into page directories when building a virtual address space. Similarly, frames are mapped into page tables.
- synchronous endpoint – Traditional IPC is performed over synchronous endpoints. Processes can send up to 484 bytes of data to each other, that they pass through the kernel via a window of their address space referred to as the IPC buffer. As the name implies, communication over a synchronous endpoint is blocking.
- thread control block (TCB) – Thread control blocks represent meta data related to threads executing in userspace. Thread-related system calls such as starting or stopping execution or modifying a thread's registers are performed using a TCB object.
- untyped memory – Untyped memory is the default state of uninitialised resources. The only possible operation on untyped memory is to split and retype it into one of the other object types described.

[1] The specific details of the target hardware platform only appear in our proofs in the initialisation specifications of Chapter 7. Even in this formalism, their only inflection to definitions and reasoning is in relation to virtual memory paging structures and the number of hardware interrupts. Though neither of these are parameterised in our theories, it would be trivial to replace the points where they appear with references to a different set of paging structures and interrupts. The lack of parameterisation in this respect matches a similar style of specificity in the seL4 formalisation.

[2] In recent versions of seL4, asynchronous endpoint objects have been renamed to notification objects. The majority of the work in this thesis was completed prior to this change. To avoid confusion stemming from inconsistent naming, we retain the legacy asynchronous endpoint terminology throughout.

On the x86 architecture, there are also additional object types for an I/O port range and an I/O address space for programming an Input/Output Memory Management Unit (IOMMU). We do not describe these in detail, as they are less relevant to the work of this thesis, but further information is available in the seL4 reference manual (Trustworthy Systems Team 2014a).

Access to an object requires a capability to that object with appropriate rights for the invocation being performed. The possible capability rights are Read, Write and Grant. The way in which they constrain usage varies across object types. For example, for synchronous and asynchronous endpoints, a Read right allows messages to be received and a Write right allows messages to be sent. Grant on a synchronous endpoint capability conveys the ability to transfer capabilities over the endpoint. Grant has no effect on an asynchronous endpoint. Kernel objects and their capabilities are typically visualised as a directed graph. An example of this can be seen in the upcoming Figure 2.6.

The capabilities of a seL4-hosted process exist in a CSpace, analogously to a program's code and data existing in a virtual address space. A CSpace is a tree[3] rooted at a CNode, which itself contains capabilities to other objects including, potentially, other CNodes. Addressing of capabilities within a CSpace is by index, referred to as a "CSpace pointer," the target of which is a "CSpace slot." The lookup of CSpace pointers works in a similar manner to page table lookups. The full algorithm for resolving the slot corresponding to a CSpace pointer is complex and detailed in full in the seL4 reference manual (Trustworthy Systems Team 2014a), but all the CSpaces we deal with in this work are single level (no root CNode contains a capability to another CNode), so resolution is straightforward.

All kernel invocations (system calls) manifest as operations on a capability that must be explicitly presented when calling the kernel. This includes IPC. A process communicates with another over a synchronous endpoint to which both processes have a capability. The sender places data in a dedicated region of its address space, known as its IPC buffer, and then invokes its capability to the shared endpoint. For example, the following code demonstrates sending a 2-word message on the endpoint to which the process has a capability in slot 14.

```
1  /* Write payload into the first two words of the IPC buffer */
2  seL4_SetMR(0, 42);
3  seL4_SetMR(1, -1);
4
5  /* Set up a descriptor for this message */
6  seL4_MessageInfo_t info = seL4_MessageInfo_new(0, 0, 0, /* length: */ 2);
7
8  /* Send the message */
9  seL4_Call(/* CSpace pointer: */ 14, info);
```

More detailed information about seL4 is available in its reference manual (Trustworthy Systems Team 2014a).

The Application Programming Interface (API) to seL4 is extremely minimal, as is typical for a microkernel. It is exposed by a small userspace library, libsel4, that mediates between the C calling convention and the kernel's Application Binary Interface (ABI). While it is possible to build strongly isolated systems on seL4 (Klein et al. 2010), the process is cumbersome and the engineering effort required is relatively large. To expose a higher level – but still general purpose – API to userspace programs requires a further abstraction. This function is served by a component platform, as described in the following section.

[3] Technically a CSpace is not necessarily a tree because it can contain cycles when CNodes contain capabilities to other CNodes. However, this only occurs in dynamic systems. In the static systems we target, CNodes never contain capabilities to CNodes.

2.2 Component-Based Software Engineering

The scale and complexity of large software systems can pose a significant engineering challenge. One of the established techniques for coping with such a challenge is Component-Based Software Engineering (CBSE) (Szyperski 1997). This methodology involves designing a system as a set of isolated functional units that communicate over well-defined, explicit channels. This separation of concerns allows individual components to be developed in isolation and then assembled to form a composite system.

A system designer typically provides two inputs when building a component-based system:

1. An architecture description that defines the component instances and how they are connected; and
2. Source code for each type of component.

These two artefacts can be visualised diagrammatically. The architecture description can be thought of as a collection of boxes representing the component instances and lines between them representing the connections. The source code can be thought of as the processes that run inside the boxes. For example, Figure 2.1 depicts a system with four component instances, A, B, C and D, with connections between A and B, A and C and B and D. The designer provides four distinct pieces of source code for this system; one for each of the component instances. Component systems are often conceptualised this way, though both these artefacts are typically specified as textual documents; the first in an Architecture Description Language (ADL), and the second in a standard programming language like C.

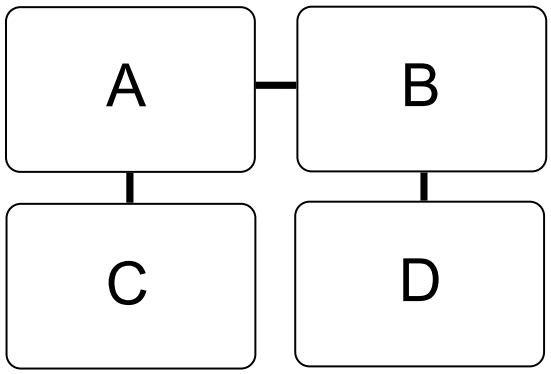

Figure 2.1: Example component system depiction

A nuance that is not typically depicted in component system visualisations is component *types*. A component type may be instantiated multiple times. Thus, in the previous example, perhaps A and B are two component *instances* of the same component *type*. In this case the designer would only provide three pieces of component source code, as these are given per type. Though the absence of this information in depictions can lead to some ambiguity, in practice it rarely causes confusion.

The heart of a component-based software environment is a component platform. This is a framework made up of an offline build environment and an online runtime environment. At build time, a code generator supplied by the platform consumes the system architecture description as input and outputs glue code that implements the connections between component instances. This glue code contains stubs that mediate between the high level functional interfaces of the component instances and the low level mechanisms provided by the operating system kernel. A program binary for each component instance is formed by compiling the provided hand written source code for the component and the generated glue code and linking the two into a final image. A component platform's development environment generally makes these steps transparent to the system

designer, leaving them to think only about the two input artefacts listed previously. This process is depicted in Figure 2.2.

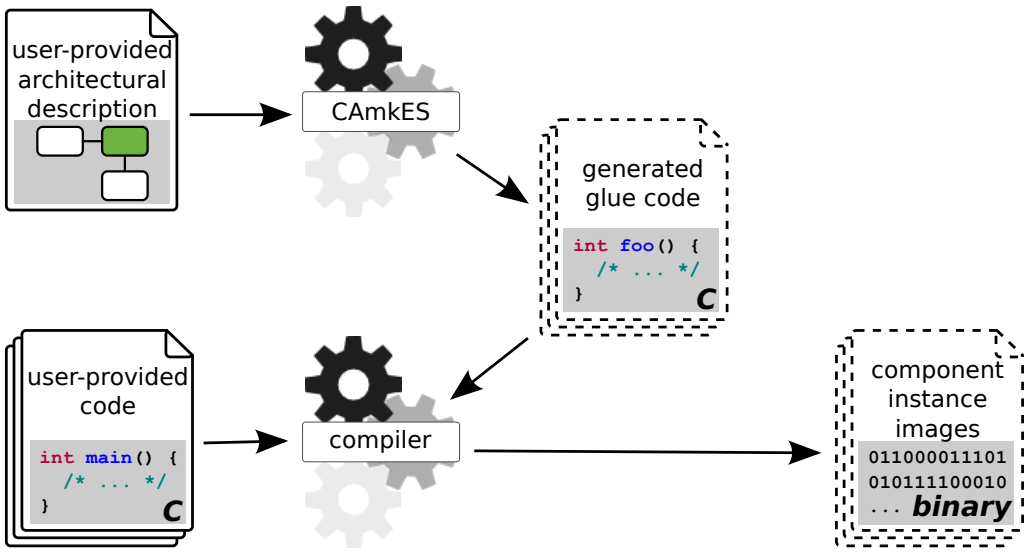

Figure 2.2: Building a component-based system

Component platforms can be categorised into dynamic and static, based on whether they allow runtime changes to the system architecture or not. The present work focuses on static component platforms as these are more suitable for high-assurance environments. Runtime reconfiguration is rarely required in such systems and a static architecture can be implemented without the need for a supporting persistent runtime process, thereby reducing the trusted computing base.

CBSE has advantages beyond simplifying the initial development of a system. By building all system-specific functionality on a component platform, the system can be easily transferred to another hardware platform or another operating system. The abstraction of the component platform promotes and facilitates this portability. The boundaries between components in a system also establish APIs for interaction between modules. This allows changes to communication patterns at the architecture level without needing to refactor component code. For example, patterns like communication interposition, filtering or virtualisation become trivial to introduce.

An aspect of CBSE that is particularly desirable when building systems for certification and verification is an explicit system architecture as an input artefact. Some analysis of system safety and security can be performed statically on this artefact (Andronick, Greenaway, and Elphinstone 2010), without the need to perform complex code validation or dynamic analysis. We show a practical instance of this in Chapter 4.

2.2.1 CAmkES

CAmkES is a component platform for microkernel-based embedded systems (Kuz et al. 2007). In terms of the distinction previously discussed between dynamic and static systems, CAmkES is a static system and has no persistent runtime process. All configuration of a CAmkES system is performed by an initial userspace loader that exits on completion. As noted, this is well aligned with high-assurance environments which typically do not rely on runtime reconfiguration. CAmkES generates communication stubs at compile time that the user can call to effect component interaction. These stubs are implemented by composing primitive operations provided by the underlying operating system kernel.

CAmkES provides three communication abstractions: remote procedure call (RPC) that is similar to the semantics of function calls, events that represent asynchronous signalling and dataports that represent shared memory. Within each of these categories, there are multiple ways to implement the given communication style, and CAmkES comes with several implementations of each, called connectors. For example, when targeting seL4 there are three commonly used RPC connectors: seL4RPCSimple, seL4RPC and seL4RPCCall. It is also possible for a system designer to implement and provide their own connectors. It might be desirable to do this to encapsulate extra boilerplate code or to use other kernel- or hardware-specific mechanisms. In the upcoming chapters, we deal with three specific standard connectors, the seL4RPCSimple RPC connector, the seL4AsynchNative event connector and the seL4SharedData dataport connector, but our work could be extended to other connectors. We expand on this possibility in Section 9.2.

CAmkES connectors are, by default, one-to-one, in the sense that they connect a single sending component instance to a single receiving component instance.[4] However, it is possible for a connector to specify that it supports multiple senders, multiple receivers or both. This allows system designers greater expressivity in implementing patterns like broadcasting or multi-client services. In the present work, we focus on one-to-one connectors for simplicity, but we note support for multiple senders and receivers here to give a more complete overview of CAmkES. Extension of this work to multiplexing connectors would require a more complex model of kernel interaction and context switching, that we extrapolate on in Section 9.1.

The CAmkES code generator has the ability to target different operating systems and programming languages, but in this work we focus on the backend for the seL4 microkernel and the C programming language. The advantage of targeting a formally verified microkernel, is that we can achieve a strong guarantee of overall system correctness by composing proofs of userspace applications with the existing proofs of the kernel's correctness. C programs can be compiled to executables suitable for a bare metal environment and thus we do not need to assume the correctness of a language runtime. We do need to assume the correctness of the translation to binary code performed by the C compiler; this is out of scope for our current work. There are other existing approaches that have the potential to address this (Leroy 2006; Sewell, Myreen, and Klein 2013).

CAmkES represents connections and their types, connectors, explicitly. This distinction is relevant as many component platforms do not (Flatt and Felleisen 1998; Levis et al. 2005). An explicit concept of connectors allows properties to be framed on connectors themselves. For example, it may be claimed that a particular connector only allows unidirectional information flow.

CAmkES supports a concept of "composite components," components that themselves contain other component subsystems.[5] The present work does not deal with composite components and requires that any input specification does not contain any such components. This is not a significant limitation as composite components are essentially syntactic sugar; any system involving composite components can be rewritten as a system without composite components. We expand on this in Appendix A.

2.2.1.1 Visualisation

In Section 2.2, we showed an example of how component systems are typically depicted. This was a generic architecture, with no details of the connections between component instances. When depicting a CAmkES system, the mode of communication of a connection is indicated by a decoration applied to its line.

RPC connections are shown with a line ending in a circle on the side of the callee (provider) and a line ending in an arc on the side of the caller (user). These ends meet, indicating the matching relationship between the

[4] In a dataport connection, the sender and the receiver are equivalent as the connection is bidirectional. However, the default mode for a dataport connector is similarly one-to-one, connecting only two component instances.

[5] Composite components are also sometimes referred to in the literature as "compound components."

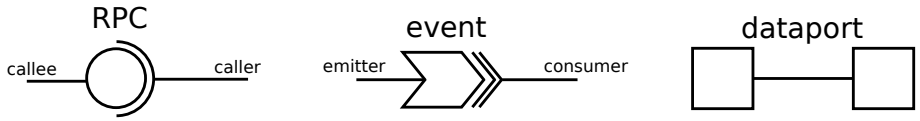

Figure 2.3: Connection mode indicators

two. Event connections have an irregular hexagon on the side of the emitter and an angled line on the side of the consumer. Dataports have a box at either end of the connecting line, representing the memory window through which each component instance accesses contained data. Figure 2.3 shows these three alternatives.

It is typical in component system architecture diagrams to omit any interface or connection names. That is, connections typically appear as unannotated instances of the alternatives in Figure 2.3. However, for the purposes of this thesis, we annotate connection lines with both the connection name and interface names for additional clarity. An example of this is demonstrated in the next section in Figure 2.4.

2.2.1.2 Syntax

As discussed, architecture descriptions of CAmkES systems are provided in a textual language. The language is declarative and C-like, with brace-delineated blocks. Procedural interfaces (endpoints for RPC) are described as a block of method declarations.

```
1 procedure my_proc {
2   void ping(void);
3   int echo(in int x);
4   int get_coordinates(out unsigned int x, inout unsigned int y);
5 }
```

This syntax is referred to as Interface Definition Language (IDL). Such blocks define a set of functions that comprise an interface and should be familiar from a programming standpoint. Where they deviate from conventional function prototypes is that each parameter has a direction. In the previous listing, the `x` parameter to `echo` is an input parameter, while the `x` parameter to `get_coordinates` is an output parameter. The `y` parameter to `get_coordinates` is used as both an input and an output.

The types of method parameters and return values in CAmkES are divided into language-independent types and C-specific types. Language-independent types such as `int` map to different native types in different target languages. Though we only deal with CAmkES' C backend in this work, the design of CAmkES permits different components within the same system to be implemented in different languages, with the ability to exchange data over a procedure method. Using C-specific types, for example `uint8_t`, in a method prevents its implementation or use in a component that is not written in C. These are provided for optimisation or when a type's in-memory representation affects correctness. When targeting a language like C that allows definitions of new types (structs), it is also possible to specify parameters of a user-defined type, the details of which are not known by the CAmkES platform. There is also an array type, whose element type can be any of the previously introduced types.

The architectural elements of the system are expressed in ADL. Component types appear as blocks that describe their interfaces and whether they have a main thread (`control`) or are passive.

```
1 component my_component {
2   control;
3   uses my_proc p;
4 }
```

The architecture of the system, as visualised by the designer, is described within an `assembly` block. This block takes the types that have been defined in the specification and describes how to compose them to form a specific system architecture. For this example, we consider a system that appears in Figure 2.4.

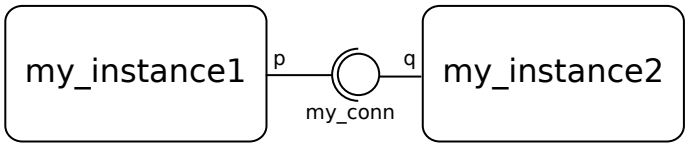

Figure 2.4: Example architecture

The `assembly` block contains a `composition` block that instantiates component types and wires them together with connections. The textual description for our example system appears as follows.

```
1 assembly {
2   composition {
3     component my_component my_instance1;
4     component another_type my_instance2;
5
6     connection seL4RPCSimple my_conn(from my_instance1.p, to my_instance2.q);
7   }
8 }
```

This is a very brief description of only the basic aspects of CAmkES' syntax. We expand on this when formalising syntactic elements in Chapter 4.

2.2.1.3 Glue Code

When building a CAmkES system, a code generator that creates the platform's environment runs prior to the compilation phase. This generated code is referred to as glue code because it conceptually "glues" the component instances together.

Recalling Figure 2.2, there are two salient points to note in this work flow diagram. Firstly, CAmkES' code generator derives the glue code purely from the architectural description of the system. That is, the user-provided code has no influence on the form of the generated code. Secondly, the glue code is compiled *with* the user-provided code. The implications of this in a language like C that lacks memory safety, are that misbehaving glue code can affect user-provided code and vice versa. This aspect complicates safety guarantees and leads to unusual challenges in the verification of component-based systems.

Taking the example from the previous section, this defines two component instances with an RPC connection linking them. The instance `my_instance1` is given an exposed function for each method in the defined procedure, under names prefixed by its interface's name: `p_ping`, `p_echo` and `p_get_coordinates`. The instance `my_instance2` is expected to implement these functions. Its definition (not shown) names its corresponding interface `q`, so its implementations are expected to carry this prefix: `q_ping`, `q_echo` and `q_get_coordinates`. At runtime, these differing symbols need to have the same semantics. Connecting the corresponding symbols seamlessly is the role of the glue code.

The glue code for an RPC connection operates by using the underlying seL4 IPC mechanism to transmit and receive method invocation data. On either side of the connection, it is responsible for serialising and deserialising data into and out of the IPC buffer. In the context of a component platform, these operations are

typically referred to as marshalling and unmarshalling, respectively. The logic for marshalling and unmarshalling method data are intended to be inverses. Informally, `unmarshal (marshal x) = x`. Figure 2.5 depicts the interaction between hand written and generated code with an abstract version of the glue code's functionality.

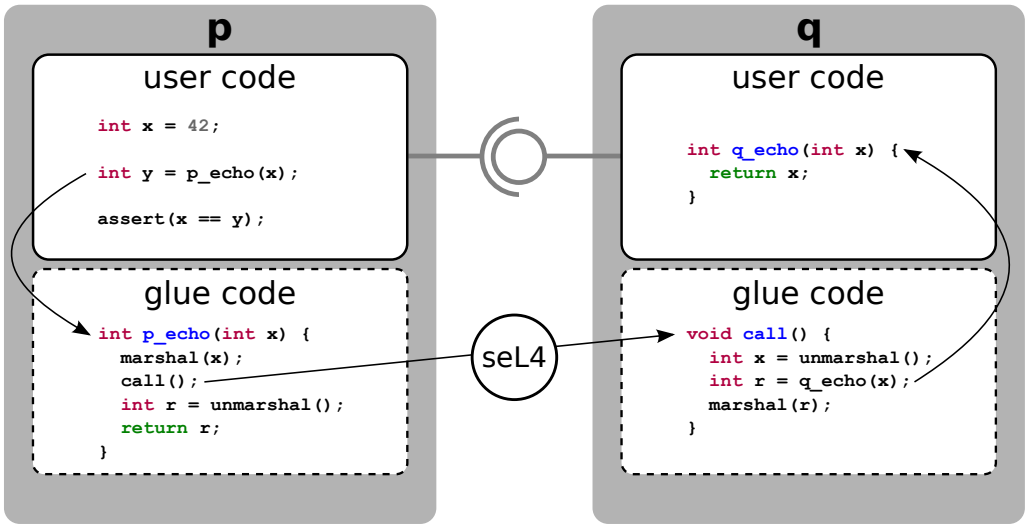

Figure 2.5: Interaction between user-provided code and glue code

Even for this simple system, the glue code produced for the seL4RPCSimple connector is extensive, so for simplicity we show just the sections related to the `echo` method. This is sufficient to give a general idea of the form and role of glue code. For `my_instance1`, the glue code for `p_echo` is as follows.

```
1  static unsigned int p_echo_marshal(int x) {
2    unsigned int _camkes_length_57 = 0;
3
4    /* Marshal the method index */
5    seL4_SetMR(_camkes_length_57, 1);
6    _camkes_length_57++;
7
8    seL4_SetMR(_camkes_length_57, (seL4_Word)x);
9    _camkes_length_57++;
10
11   return _camkes_length_57;
12 }
13
14 static void p_echo_call(unsigned int length) {
15   /* Call the seL4 endpoint */
16   seL4_MessageInfo_t info = seL4_MessageInfo_new(0, 0, 0, length);
17   (void)seL4_Call(5, info);
18 }
19
20 static int p_echo_unmarshal(void) {
21   unsigned int _camkes_mr_58 = 0;
22
23   int _camkes_ret_59 = (int)seL4_GetMR(_camkes_mr_58);
24   _camkes_mr_58++;
25
26   return _camkes_ret_59;
27 }
28
```

```
29 int p_echo(int x) {
30   /* Save any pending reply cap, as we'll overwrite it during `seL4_Call` */
31   camkes_protect_reply_cap();
32
33   /* Marshal input parameters */
34   unsigned int _camkes_mr_index_60 = p_echo_marshal(x);
35
36   /* Call the endpoint */
37   p_echo_call(_camkes_mr_index_60);
38
39   /* Unmarshal the response */
40   int _camkes_ret_61 = p_echo_unmarshal();
41
42   return _camkes_ret_61;
43 }
```

This code serialises the function arguments into the process' IPC buffer (lines 1-12) and then calls the kernel to transfer this payload (lines 14-18). On return, it deserialises the response from the callee (lines 20-27) and returns to the user. It works in cooperation with the glue code for `my_instance2`, the relevant section of which is as follows, that we will explain following its listing.

```
1  /* Implementation provided by the user */
2  extern int q_echo(int x);
3
4  /* Per-thread globals */
5  static int echo_x_1;
6  static int echo_x_2;
7
8  static int *get_echo_x(void) {
9    switch (camkes_get_tls()->thread_index) {
10     case 1:
11       return &echo_x_1;
12     case 2:
13       return &echo_x_2;
14     default:
15       assert(!"invalid thread index");
16       abort();
17   }
18 }
19
20 static void q_echo_unmarshal(int *x) {
21   unsigned int _camkes_mr_58 = 1; /* 0 contained the method index */
22
23   *x = seL4_GetMR(_camkes_mr_58);
24   _camkes_mr_58++;
25 }
26
27 static int q_echo_invoke(int x) {
28   /* Call the implementation */
29   return q_echo(x);
30 }
31
32 static unsigned int q_echo_marshal(int _camkes_ret_59) {
33   unsigned int _camkes_mr_60 = 0;
34
35   seL4_SetMR(_camkes_mr_60, (seL4_Word)_camkes_ret_59);
36   _camkes_mr_60++;
```

```
37
38    return _camkes_mr_60;
39  }
40
41  static unsigned int q_echo_internal(void) {
42    int *x = get_echo_x();
43    q_echo_unmarshal(x);
44
45    int _camkes_ret_61 = q_echo_invoke(*x);
46
47    unsigned int _camkes_length_62 = q_echo_marshal(_camkes_ret_61);
48
49    return _camkes_length_62;
50  }
51
52  static seL4_MessageInfo_t q__run_internal(bool _camkes_first_69,
53      seL4_MessageInfo_t _camkes_info_68) {
54    if (_camkes_first_69) {
55      _camkes_info_68 = seL4_Wait(6, NULL);
56    }
57
58    /* We should have at least been passed a method index */
59    assert(seL4_MessageInfo_get_length(_camkes_info_68) >= 1);
60
61    int _camkes_call_70 = (int)seL4_GetMR(0);
62
63    switch (_camkes_call_70) {
64
65      /* ... */
66
67      case 1: { /* echo */
68        unsigned int _camkes_length_72 = q_echo_internal();
69
70        /* Send the response */
71        _camkes_info_68 = seL4_MessageInfo_new(0, 0, 0, _camkes_length_72);
72
73        seL4_ReplyWait(6, _camkes_info_68, NULL);
74
75        break;
76      }
77
78      /* ... */
79
80    }
81
82    return _camkes_info_68;
83  }
84
85  int q__run(void) {
86    bool _camkes_first_74 = true;
87
88    seL4_MessageInfo_t _camkes_info_75 = { { 0 } };
89    while (1) {
90      _camkes_info_75 = q__run_internal(_camkes_first_74, _camkes_info_75);
91      _camkes_first_74 = false;
92    }
93    return 0;
94  }
```

This code deserialises a message from the caller (lines 20-25), invokes the user-provided `echo` implementation (lines 27-30), then serialises (lines 32-39) and sends the return value back. Note that the code is driven by an outer event loop (lines 85-94) that repeatedly invokes a dispatch function (lines 52-83) that discriminates by an index indicating which method the caller invoked. The wire format of messages can be inferred from these two pieces of glue code, but it is not strictly relevant to the system designer as long as it is agreed on by the two pieces of glue code. The receiving glue code has some additional code for thread-local variable support (lines 4-18). This is driven by constraints of our verification toolchain and does not impact usage by a system designer.

The first piece of glue code is compiled with the user-provided code for `my_component` to form one binary for `my_instance1`, while the second piece of glue code is compiled with the user-provided code for `another_type` to form a binary for `my_instance2`. Note how these two pieces of glue code collaborate to form the component platform abstraction, namely that `my_instance1` can directly call the `echo` implementation in `my_instance2`. For example, `my_component` could contain the following sequence of statements.

```
1  int x = 42;
2
3  int y = p_echo(x);
4
5  assert(x == y);
```

This would be paired with a corresponding implementation in the code of `another_type`.

```
1  int q_echo(int x) {
2    return x;
3  }
```

It may seem that the glue code is overly verbose and omits some obvious optimisation opportunities. This is not atypical of static component platforms. The glue code is deliberately straightforward and transparent in order to provide the C compiler a greater ability to optimise it during compilation. The only optimisations CAmkES performs during code generation are those that cannot be performed by the C compiler due to information it is lacking. The combined effect of CAmkES' optimisations and the C compiler's optimisations effectively reduce the cost of the component platform's abstractions to zero. That is, the introduced glue code is at least as fast as equivalent hand written code.

2.2.1.4 Hardware Devices

Hardware devices are represented as explicit components in CAmkES systems. The abstractions of CAmkES are well-suited to the communication modes exhibited by hardware. Memory-mapped device registers are presented as dataports, interrupts as emitted events and I/O ports[6] as provided procedural interfaces. In addition to allowing a unified way of interacting with hardware and software elements, this design assists the development of device drivers and enforces a well-defined hardware API that permits natural virtualisation of devices.

The formalisms we present in this thesis support only software components, but this section is included for completeness. Extension of our work to hardware components would necessitate interaction with a model of device behaviour.

[6] I/O ports are a hardware access mechanism not present on all platforms, so the reader may be unfamiliar with them. They are typically presented by the device to the operating system as a set of virtual registers of word or sub-word width. They are read from or written to with dedicated assembly instructions. The seL4 microkernel supports access to such ports on the x86 architecture. The access provided by seL4 is a thin security wrapper over the native x86 instructions.

2.2.1.5 Attributes

CAmkES provides a mechanism for specialising component instantiations called attributes. This serves to parameterise a component type with values set at compile time by the system designer. Attributes are defined with the following syntax.

```
component Client {
  /* An integer attribute with the name `my_id` */
  attribute int my_id;

  /* ... */
```

Within the `assembly` block that defines the system architecture, the attribute value for each component instance can be set within a `configuration` block. For example, if we have two instances of the above component, they can each be given a different value for the `my_id` attribute.

```
assembly {
  composition {
    component Client c1;
    component Client c2;
    component Server s;

    connection seL4RPCSimple conn1(from c1.inf, to s.inf1);
    connection seL4RPCSimple conn2(from c2.inf, to s.inf2);
  }
  configuration {
    c1.my_id = 1;
    c2.my_id = 2;
  }
}
```

At runtime, the attribute values are available to component code with a matching symbol name. In the example provided, the component code would have an integer variable, `my_id`, that would have the value `1` in the first component instance and `2` in the second.

Attributes can have the type `string` or `int`, which has so far proved sufficient for system designers. The representation of these types at runtime is dependent on the target language.

2.2.1.6 CapDL

Section 2.1.2 detailed how all authority in seL4 is conveyed by capabilities to kernel objects. This makes it possible to express the state of any seL4-based system as a collection of kernel objects and a directed graph of capabilities that reference these objects. This is formalised in the CapDL language (Kuz et al. 2010). A generic userspace boot loader accepts such a CapDL specification as input and configures a seL4 environment to match this specification (Boyton et al. 2014).

The CapDL language itself is another C-like declarative language. Specifications are composed of two sections, `objects` and `caps`, describing kernel objects and capabilities to them, respectively. An example specification is given below.

```
objects {
  my_ep = ep /* A synchronous endpoint */

  /* Two thread control blocks */
  tcb1 = tcb
  tcb2 = tcb

  /* Four frames of physical memory */
  frame1 = frame (4k)
  frame2 = frame (4k)
  frame3 = frame (4k)
  frame4 = frame (4k)

  /* Two page tables */
  pt1 = pt
  pt2 = pt

  /* Two page directories */
  pd1 = pd
  pd2 = pd

  /* Two capability nodes */
  cnode1 = cnode (2 bits)
  cnode2 = cnode (3 bits)
}
caps {
  cnode1 {
    0x1: frame1 (RW) /* read/write */
    0x2: my_ep (R)   /* read-only */
  }
  cnode2 {
    0x1: my_ep (W)   /* write-only */
  }
  tcb1 {
    vspace: pd1
    ipc_buffer_slot: frame1
    cspace: cnode1
  }
  pd1 {
    0x10: pt1
  }
  pt1 {
    0x8: frame1 (RW)
    0x9: frame2 (R)
  }
  tcb2 {
    vspace: pd2
    ipc_buffer_slot: frame3
    cspace: cnode2
  }
  pd2 {
    0x10: pt2
  }
  pt2 {
    0x10: frame3 (RW)
    0x12: frame4 (R)
  }
}
```

The capabilities described in a specification form a directed graph. The corresponding graph for the above specification is depicted in Figure 2.6, wherein each capability slot is represented as a square, shaded for occupied or unshaded for empty. Considering `cnode1` from the specification, we can see the two capabilities it contains (lines 27-30) shown as arrows to the objects `frame1` and `my_ep`. More detailed information about the CapDL syntax can be found in (Lewis et al. 2014).

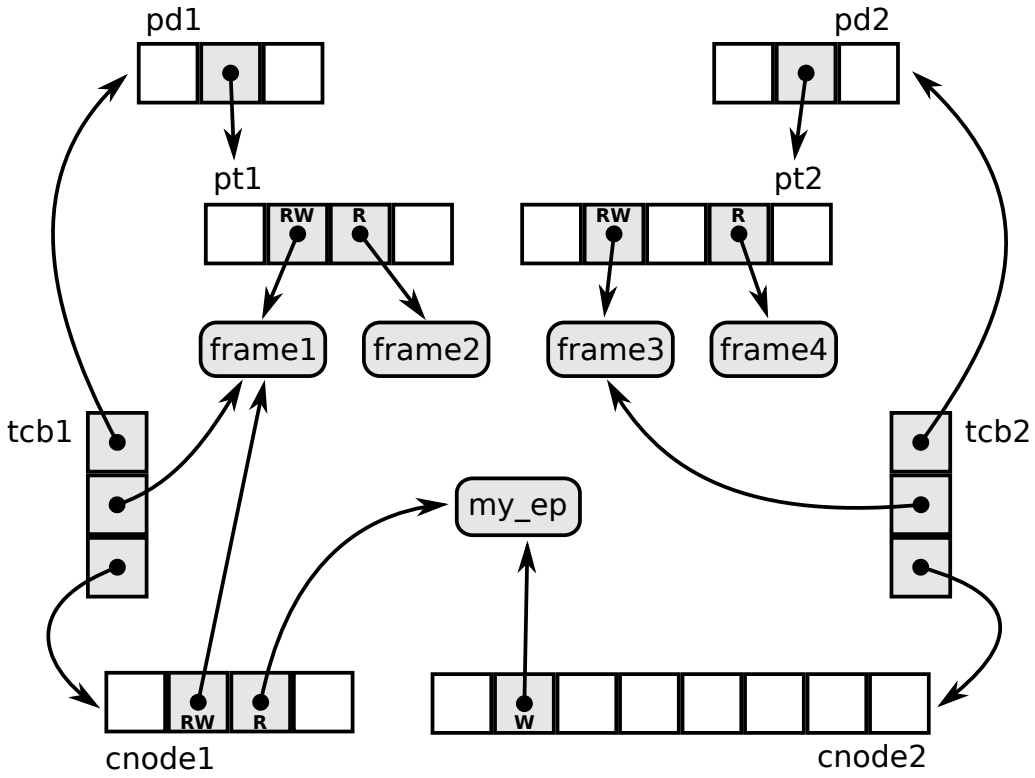

Figure 2.6: CapDL specification depicted as a graph

The CapDL language and this boot loader are fully generic and independent of the targeted system, but they become relevant in the context of CAmkES because they provide a trustworthy path to booting a CAmkES system. Instead of providing its own runtime, the CAmkES code generator produces a CapDL specification based on the designer's architecture description. Extending Figure 2.2, we depict CAmkES' additional production of an initialisation specification in Figure 2.7. By targeting a verified initialisation path (Boyton et al. 2013), one of the key challenges of component platform correctness – guaranteeing correspondence between the input architecture and the running system – is reduced to proving the correctness of the generated CapDL. This is explained in detail in Chapter 7.

2.3 Interactive Theorem Proving and Code Verification

Formal software verification is a broad field that encompasses numerous techniques for providing evidence of the correctness of programs. These include automated techniques such as static analysis tools like Coverity (Engler et al. 2000; Hallem et al. 2002) and manual techniques such as interactive theorem proving. Though manual techniques can be more time intensive than automated techniques, there are classes of problems for which no known automated solution exists, such as general functional correctness. Thus manual techniques are often used to prove stronger properties than would be possible with automated techniques. Proving general functional correctness in an interactive theorem prover also has the added advantage of avoiding con-

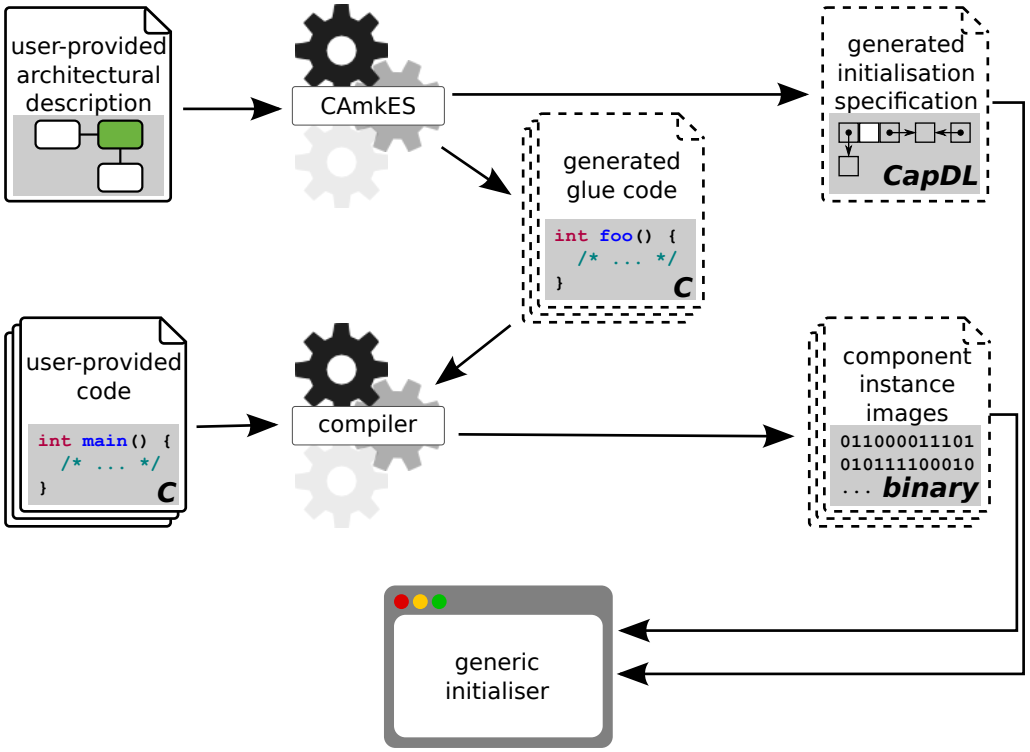

Figure 2.7: CapDL production

straining verification to a specific property. A later proof of a specific functional property can be achieved by appealing to general functional correctness.

Verifying a program via interactive theorem proving involves writing an abstract specification for the program in a formal environment. In this work, we focus on the Isabelle/HOL theorem prover (Nipkow, Paulson, and Wenzel 2002). We know of no technical limitations to our work that would prevent its implementation in another theorem prover, at least in principle. By proving properties about the abstract specification of a program, it is possible to show the correctness according to some desired behaviour.

There is a semantic gap between the abstract specification of a program and its implementation. That is, an abstract specification might describe behaviour that does not precisely correspond to how the actual program works. This gap is depicted as the arrow labelled "A" in Figure 2.8. To minimise the risk of deviation between the specification and the implementation, a model of the implementation can be constructed, either hand written or generated, and a proof that demonstrates this model refines the original abstract specification can be provided (Back 1978; Morgan 1990). This intermediate artefact is depicted as the central box and its refinement proof as the arrow labelled "B." While this process does not remove the semantic gap between formalism and implementation, it reduces the gap to that between the implementation model and the implementation itself. It is not possible to completely remove this residual semantic gap. There always remains a distance between a model and the real world. For example, hardware devices are fabricated from precise specifications, but it it is still possible for the physical implementation of a device to deviate from its specification. In the limit, a formal model is unavoidably limited by our incomplete representation of the physical world. The remaining semantic gap appears in Figure 2.8 as the arrow labelled "C."

When reasoning about code in this thesis, we focus on C code conforming to the C99 standard (ISO/IEC 2005). As discussed in Section 2.2.1, C serves as a suitable environment for high-assurance embedded systems. To accomplish this reasoning, we use the pre-existing toolchain described in the following sections.

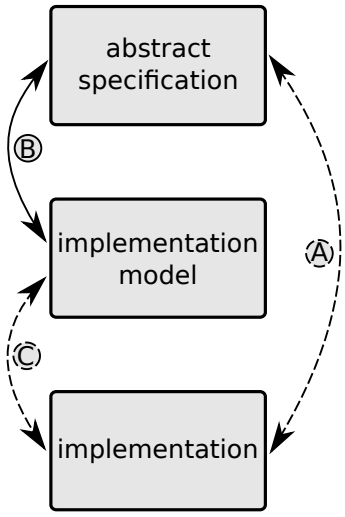

Figure 2.8: Program refinement as a relationship between artefacts

2.3.1 C-to-SIMPL Parser

A necessary ingredient for reasoning about C code is a formal semantics of the language. Such a semantics for C99 was previously developed in HOL (Norrish 1998). The accompanying parser for this work translates C code into a specification in the SIMPL language for Isabelle/HOL. SIMPL is a generic language for representing imperative programs with exceptions. The tool supports a subset of C that has been shown sufficient for implementing low level systems (Klein et al. 2009), and that is a superset of the features used by generated code in CAmkES.

As an example of the functional representation it provides, consider the following C code for taking the absolute value of an integer.

```
int abs(int x) {
  if (x < 0)
    return -x;
  return x;
}
```

The C-to-SIMPL parser transforms this definition into the following Isabelle/HOL representation of the same function.

```
definition
  abs :: "(globals myvars, int, strictc_errortype) com"
where
  "abs ≡
    TRY
      IF  x <s 0 THEN
        Guard SignedArithmetic {|- 2147483648 ≤ - sint  x ∧
                                  - sint  x ≤ 2147483647|}
        (creturn global_exn_var_'_update ret__int_'_update
          (λs. - x_' s))
      FI;;
      creturn global_exn_var_'_update ret__int_'_update x_';;
      Guard DontReach {} SKIP
    CATCH SKIP
    END"
```

As can be seen, even for a simple C function, the SIMPL representation is more verbose than one might expect. The intention of the translation is to be as faithful as possible to the semantics of C99. For example, the definition contains a guard that the negation of the input value must lie between `-2147483648` and `2147483647`, the least and greatest representable signed 32-bit numbers, respectively. This stems from §6.5.5 of the C99 standard (ISO/IEC 2005):

> *If an exceptional condition occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.*

While this may seem obvious, note that it precludes the input `-2147483648`. The negation of this value is `2147483648`, which falls outside the range of representable signed 32-bit integers and hence results in undefined behaviour. Functional representations of this verbose form can be reasoned about and strong correctness properties can be derived, however the effort to do so is magnified by having to manipulate the large terms involved.

2.3.2 AutoCorres

While the verbosity of the representations of C code produced by the C-to-SIMPL parser is a positive attribute for fidelity to the original code, it impedes reasoning by forcing a proof to contain manual steps pertaining to, for example, safe type casts.

A tool called AutoCorres (Greenaway 2015) addresses these issues by performing a set of further, verified abstractions on the SIMPL representation of C functions. The transformations of AutoCorres have been proven correct in Isabelle/HOL and therefore do not increase the amount of trusted code when interactively reasoning about a C program.

Applying AutoCorres to the previous function yields the following definition:

```
definition
  abs :: "32 signed word ⇒ lifted_globals ⇒
    (32 signed word × lifted_globals) set × bool"
where
  "abs x ≡
    condition (λs. x <s 0)
      (do guard (λs. - INT_MAX - 1 ≤ - sint x);
          guard (λs. - sint x ≤ INT_MAX);
          return (- x)
       od)
    (return x)"
```

Observe that the structure of this definition more closely matches the C definition of the function. This form also aids reasoning by automatically removing some extraneous guards; in this case a guard for reaching the end of the function.

2.3.3 Proving Functional Correctness

To state the correctness of a C function produced as an Isabelle/HOL term by AutoCorres, we employ a syntax that is a variant of Hoare triples to express the precondition and postcondition of a potentially nondeterministic monadic function (Cock, Klein, and Sewell 2008). The representations delivered by AutoCorres

are capable of expressing state modification, a return value and, if relevant, failure. The following notation states that, if the precondition `P` holds and the function `f` terminates normally, then the postcondition `Q` will hold after `f` has executed.

```
"⦃P⦄ f ⦃Q⦄"
```

`P` is a predicate over the initial state (memory, global structures, etc.):

```
"P :: lifted_globals ⇒ bool"
```

`Q` is a predicate over two parameters: the return value of `f` and the final state. For example, if `f` returned a value of type `int`, the type of `Q` would be:

```
"Q :: int ⇒ lifted_globals ⇒ bool"
```

The previous Hoare triple allows for the possibility that `f` fails or does not terminate, for example, by performing an operation with undefined behaviour in C. To express total correctness of `f`, we use the following variant that requires termination and absence of failure, including guard violations.

```
"⦃P⦄ f ⦃Q⦄!"
```

A common idiom used with this syntax is universal quantification of a state parameter to express how a function modifies the state of the system. To claim that a function modifies the state according to a specification definition, g, we write:

```
"∀s0. ⦃λs. s = s0⦄ f ⦃λ_ s. s = g s0⦄!"
```

In the case of a pure function, its modification to the state is simply the identity and we simplify the above expression to:

```
"∀s0. ⦃λs. s = s0⦄ f ⦃λ_ s. s = s0⦄!"
```

Further details of this Hoare calculus for monadic functions are available in (Cock, Klein, and Sewell 2008).

To see how this syntax is used with respect to real functions, we return to our `abs` example from the previous section. It is possible to phrase a Hoare triple describing its correctness using the syntax we have just introduced.

```
lemma abs:
  "∀s0. ⦃λs. s = s0 ∧ sint x ≠ INT_MIN⦄
          abs x
        ⦃λr s. s = s0 ∧ r = (if x <s 0 then -x else x)⦄!"
```

This statement claims that any input that is not `INT_MIN`, a constant for the largest representable negative integer, will yield an output that is the negation of the input. This is a case in which we use the universally quantified `s0` to express that this function is pure.

The proof of this lemma has been elided, but it consists of five straightforward steps. For simple examples such as this, existing automated tactics (Winwood et al. 2009) make proofs straightforward. Having defined a leaf function and proven its semantics, we can reuse this lemma in proofs of its callers. Consider the following caller, a function that divides the absolute value of two integers.

```
int abs_div(int n, int d) {
  return abs(n) / abs(d);
}
```

We can state a semantics for this function in a similar style to `abs`.

```
lemma abs_div:
  "∀s0. {λs. s = s0 ∧ sint n ≠ INT_MIN ∧ sint d ≠ INT_MIN ∧ sint d ≠ 0}
          abs_div n d
        {λr s. s = s0 ∧ r = (if n <s 0 ∧ ¬ d <s 0 ∨ ¬ n <s 0 ∧ d <s 0
                               then - (n sdiv d)
                               else n sdiv d)}!"
```

This lemma states that, if neither of the inputs are `INT_MIN` and `d` is not zero, the result is the expected quotient. It can be proven by reusing the previous lemma of the correctness of `abs`. The proof itself, that has been elided, consists of forty-nine uncomplicated, yet laborious steps. This already makes a strong case for automation where possible. The work in this thesis builds on this formalism and demonstrates how to automate such proofs when the functions they reference are sufficiently regular.

2.4 Correctness

One of the stated goals of this thesis is to guarantee the correctness of a component platform. However, the notion of "correctness" is not necessarily well-defined with respect to component platforms. While Section 2.3 demonstrated intuitive and obvious correctness criteria for a simple, isolated function, correctness for a general purpose abstraction like a component platform is more slippery.

In this section, we elaborate on what properties are necessary and sufficient to guarantee the correctness of a component platform, and how the correctness of a component platform influences the correctness of component-based systems. We hope this serves to clarify our goals and orient the reader to our work. We explore notions of correctness in a top-down manner, beginning with the notion of global correctness in a component-based system. Note that this section expands on discussion from previously published work (Fernandez, Andronick, et al. 2015a).

2.4.1 Global Correctness

Correctness, as a property of a large software system, is a requirement of the system as a whole. For example, consider a component-based aviation system responsible for collision avoidance. It may be said to be correct if its operation always results in safe distances between aircraft. This requires correct cooperation between the components performing calculations and providing feedback, the component platform mediating between the components, and the underlying operating system kernel. This decomposition of global correctness is represented visually in Figure 2.9.

We deal with the correctness of each of these elements individually in the following sections. It is inherent in the approach of CBSE that each hosted element depends on its hosting environment. That is, the correctness of the components in a system is predicated on the correctness of the component platform, which is in turn predicated on the correctness of the underlying operating system kernel. A failure in any layer of the system invalidates the guarantees of those on top.

To say that a component-based system is correct, we need guarantees for the correctness of each of these three elements in isolation, and we need these guarantees to be *composable*. That is, the formal properties of each

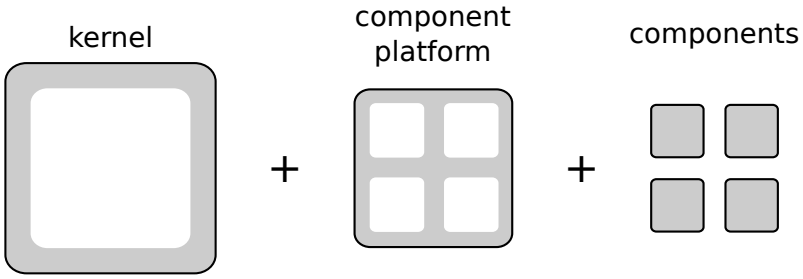

Figure 2.9: Compositional correctness

of these artefacts that are shown in isolation, must remain valid in a system formed by their composition. Ideally these properties are formally connected in a chain, with the assumptions of one formalism discharged by the proven properties of the previous. For example, the component platform makes assumptions on the kernel mechanisms it uses, and it is desirable to prove these assumptions as part of the correctness of the kernel itself. To decompose global correctness, we first consider the correctness of each of these elements in isolation.

2.4.2 Kernel Correctness

While correctness of the operating system is not one of the direct targets of this work, it is informative to define what properties we assume the operating system kernel provides. Our component platform implementation targets seL4, and we rely on its functional correctness (Klein et al. 2009) and strongly-guaranteed, hardware-enforced isolation mechanisms (Sewell et al. 2011). The generated code of the component platform is formed by composing operating system primitives to effect communication between component instances. These primitives are called through a userspace library that provides a thin wrapper that mediates between the C calling convention and the kernel's ABI. While we provide functional correctness proofs for some stub functions from seL4's supporting userspace library, we axiomatise the semantics of the system calls themselves.

We only use a subset of the kernel's API and the semantics we assume are not overly complex to state (for example, the `seL4_Call` system call, when provided with a valid capability to a synchronous endpoint, transfers a requested amount of data to a receiver). However, the proof of these semantics is beyond the scope of this thesis. We expand on this limitation in Section 9.1.

While our work explicitly targets seL4, there is no fundamental dependency on this specific kernel in the techniques we apply. Our approach could be applied to equal effect on another kernel that provides equivalent guarantees, though we are not aware of any currently in existence. It would also be possible to apply our approach on top of a kernel that provides weaker guarantees. However, as the guarantees of the component platform are predicated on kernel correctness, this would weaken the global assurance property.

2.4.3 Platform Correctness

Correctness of a component platform, while seemingly intuitive, is challenging to formally define. It is straightforward to observe that a component platform should provide isolated containers for component instances to run in and faithful communication mechanisms for architectural connections, but this is imprecise and it is not easy to formulate exactly what we mean by the "isolation" and "communication" provided by the platform.

The idea of guaranteeing correctness of a component platform is not new, and there is a wide range of existing work targeted at this problem (Basu et al. 2011; Cobleigh, Giannakopoulou, and Păsăreanu 2003). However,

the majority of existing work targets one or both of two related but divergent issues: the correctness of component interaction or the compatibility of interfaces. The former of these is concerned with the correctness of *components*, rather than the *platform*, and is more closely aligned with the next section. The latter is a correctness property of the platform defined by its ability to ensure component instances are only ever connected in semantically consistent ways. While this is necessary for correctness, it is insufficient by itself. It is also unclear how work in either area could be formally connected to, and integrated into, a larger correctness argument encompassing the entire platform and operating system kernel.

Conflating issues related to the correctness of component-based systems relegates the correctness of the platform itself to an implicitly defined concept and conceals important concerns. For example, no existing work of which we are aware guarantees that the code generated by a component platform does what it claims without other side effects. This is not intended as an indictment of existing component platform formalisms. The code guarantees we allude to are fragile in isolation and too vulnerable to invalidation from below by bugs in the operating system primitives they rely upon. For example, given a formal proof of correctness of generated RPC code, this proof is predicated on the assumption that the operating system's primitives for transmitting byte data are correct. Such a proof can be trivially undermined by a bug in the operating system implementation that affects the semantics of this primitive.

In the past there has been little value in providing such guarantees predicated on an unstable foundation. However, we are now in a position to better anchor such formalisms. By building on a formally verified operating system kernel, these guarantees take on more substance and derive their robustness by extending the well-founded properties of the kernel. It remains as a challenge for us then to define a provable and useful correctness property for a component platform.

The key insight that led us to our final definition of correctness was that a component platform serves as a mental abstraction for the system designer. Its utility to them is to allow them to think of their component-based system in terms of component instances and their interactions, without regard for cross-cutting complexities or the implementation of the "boxes and lines" in their conceptual view of the system. This observation leads to a natural and intuitive notion of a component platform being correct if its implementation preserves the semantics of its abstractions. The parallels to program refinement here are not entirely coincidental.

Narrowing this informal definition to a more precise expression, we define the correctness of platform-generated code by its preservation of the functional semantics of the communication abstraction it implements. For RPC glue code, an invocation from one component instance to another should be indistinguishable from a direct function call between the corresponding two. That is, a *remote* procedure call should be functionally equivalent to a *local* procedure call. This property has an elegant equivalent formulation in the functional correctness of generated code (which we elaborate on in Chapter 6), and it corresponds well with the system designer's intuition of what the component platform's function should be. It guarantees that the RPC glue code "does what it says on the box."

To quantify the isolation guarantees provided by a correct component platform, we have chosen to leverage the capability-based access control in seL4 and the accompanying access control formalisation (Sewell et al. 2011). By generating a CapDL specification of a component system, as described in Section 2.2.1.6, we have a formal representation of the authority within the system, that prior work has guaranteed to accurately reflect the system at runtime (Boyton et al. 2013). We define an independent model for representing a CAmkES system (Chapter 4), capturing the precise structure of a component-based system architecture. To unify the two models, we provide a mapping between CAmkES specifications and CapDL specifications, for which we generate a specialised proof on a per system basis (Chapter 7). Recalling Figure 2.7, we now extend this diagram with the production of a correspondence proof in Figure 2.10. The outcome is a formal link between a component-based system architecture and a capability specification, from which an access control policy

can be automatically inferred. It then becomes possible to formally prove precise upper bounds on the effect one component instance can have on another, while maintaining a strong link between the model and the implementation. This technique for guaranteeing isolation mechanisms effectively lifts the isolation properties of seL4 to the level of abstraction of the component system architecture, and is expanded upon in Chapter 7. Again, while we use seL4-specific mechanisms to accomplish our aim, the techniques generalise to any capability-based system.

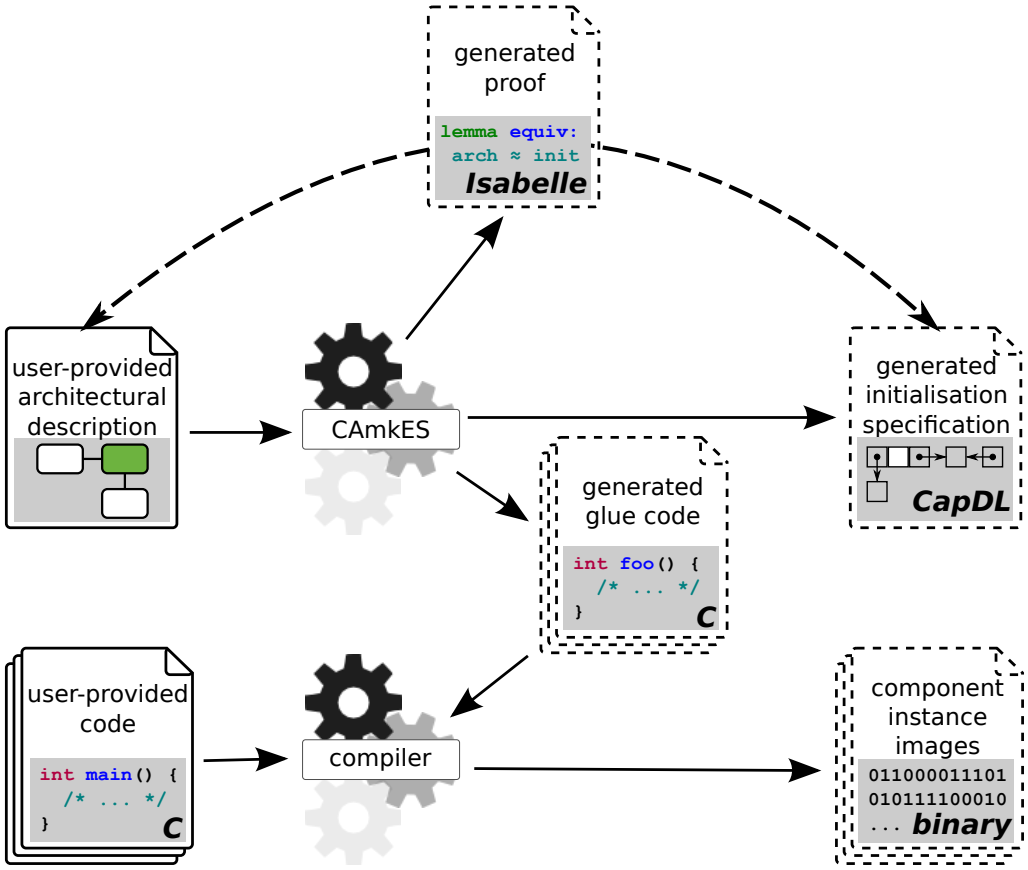

Figure 2.10: Initialisation proof production

2.4.4 Component Correctness

It may not be immediately obvious, even to practitioners, that the correctness of a system of connected component instances is not a general purpose property. That is, what is "correct" for a component-based system is intimately dependent on the system's purpose. Recalling the aviation collision avoidance system from Section 2.4.1, the correctness criteria for this system is fundamentally inseparable from the specific intended goal of that system. For this reason, unlike an operating system kernel or a component platform, it is not possible to provide a generalised proof of the correctness of a system of component instances. Instead, the approach we take in this thesis is to improve the state of the art in mechanisms and tools available for system designers to prove the correctness of their systems.

As discussed previously in this chapter, component correctness is predicated on the correctness of the component platform hosting the component-based system. For this reason it is crucial to expose a way to integrate proofs of component correctness with generated artefacts that guarantee the platform's correctness. The gradated techniques we show for doing this in Chapter 4, Chapter 5 and Chapter 6 go beyond what is present in

existing work by integrating seamlessly with correctness guarantees of the component platform. Where existing frameworks for reasoning about component-based systems bottom out in (often implicit) assumptions about the component platform, we bridge this gap, providing a formal link to guarantees of the platform's correctness.

2.4.5 Composable Correctness

Informal associations between formalisms promote ambiguity and leave room for hidden assumptions or semantic mismatch. In a proof chain, the risk is that the assumptions of one artefact may not be satisfied by the guarantees of the previous. Such mismatches can be subtle and easily obscured by disconnected type systems.

To minimise the likelihood of such inconsistencies, we pursue a formal chain from the correctness proofs of the kernel to the correctness proofs of specific component compositions. At the junction between the component platform and the kernel, we tackle this problem by phrasing the properties of the platform in the same formalism as the seL4 proofs. For code proofs, this means phrasing our proofs in the same generic model of the C99 standard as the kernel proofs (Norrish 1998) and referring to specifications of the kernel's primitives (system calls) as the only external functions. There remains an informal connection with respect to the kernel's userspace stubs and their semantics, that we detail in Section 9.1. Using the same approach for the isolation guarantees, we build on the existing CapDL formalism so we are once again "speaking the same language" as the kernel's existing proofs.

To draw a formal connection between the component platform and a specific component system, we allow the system designer full visibility of the component platform proofs. By directly exposing the platform proofs for reuse by the system designer we admit a proof without assumptions on the platform. More specifically, the system designer can phrase a lemma about their own code which calls platform-generated code, and have lemmas about this generated code available for use within their proof without incurring additional assumptions. By obviating the need for further assumptions, we allow them to achieve an assurance level as if they had hand written the generated code and manually proven its correctness. The practical robustness of this integration is that it is not possible for the system designer to inadvertently assume a property that the generated code does not provide.

2.5 Summary

This chapter has introduced the basic building blocks upon which our present work is founded. By providing an overview of microkernels, component-based software engineering and interactive theorem proving, we intend to have oriented the reader for the material that is to come. When reading the remainder of this thesis, the present chapter should be referred back to for clarification where necessary.

By narrowing the definition of correctness with respect to component platforms, we have aimed to clarify precisely what we mean by our claim of a verified component platform. It is our hope that this nuance is dealt with in a similarly explicit manner in future verification efforts for component-based systems.

Chapter 3

Related Work

The approach taken in this thesis builds on a foundation of previous theory and spans several areas of research. In particular, we combine component-based software engineering and formal methods to enable robust, reusable software. Previous work has identified this combination as the most promising approach to building critical software at scale (Meyer 1997; Meyer 2003), and our work extends the state of the art in this domain.

In this chapter, we present a discussion of the main areas of related work and note how they form the context in which our work is situated. While a treatment of the most closely related research fields is consolidated here, individual pieces of work are referenced in other chapters when they are most relevant to particular sections.

3.1 Component Platforms

While the focus of this thesis is not the design and implementation of a component platform itself, it is worthwhile identifying some other similar platforms to CAmkES in order to characterise its feature set. We hope this may serve to indicate the portability and applicability of our work to other mainstream component platforms. It is difficult to cover the vast array of component platforms in a meaningful discussion, so we choose instead to highlight a select set that are widely deployed and/or close in implementation to CAmkES.

Perhaps the most familiar component platform to programmers is CORBA (*Common Object Request Broker Architecture 3.3 Specification* 2012; *Common Object Request Broker Architecture Components* 1999). The name, CORBA, is used to describe a broad ecosystem for building component-based systems, which leads to some ambiguity, but it was one of the first platforms to popularise many of the concepts that are ubiquitous in modern CBSE. CORBA has a well-designed IDL that has been reused in other component platforms (for example, Spring (Mitchell et al. 1994)) and has served as the basis for CAmkES' IDL. Targeted primarily at C/C++ programmers, the motivations for its syntax are described in the first version of the specification (*Common Object Request Broker Architecture Specification, Version 1.0* 1991):

> *IDL obeys the same lexical rules as C++, although new keywords are introduced to support distribution concepts. It also provides full support for standard C++ preprocessing features... IDL grammar is a subset of ANSI C++ with additional constructs to support the remote operation invocation mechanism. IDL is a declarative language; it supports C++ syntax for constant, type, and operation declarations; it does not include any algorithmic structures or variables.*

The IDL syntax is an unusual point of similarity between CORBA and CAmkES, and in most other aspects they differ. CORBA is a large, all-inclusive platform. It defines a specification,[1] allowing multiple implementations, covering several axes of dynamism and a complex type system. By this comparison, CAmkES is a deliberately conservative component platform, with constraints driven by high-assurance target environments.

Many component platforms make a distinction between the languages they use to describe interfaces and architectures; for example (Cervantes and Hall 2004; Ommering et al. 2000). While CAmkES was originally designed this way, its IDL and ADL have since been unified, allowing users to intermingle the two syntaxes. The distinction still appears in our formal architecture model (Chapter 4) and the syntax is not as tightly integrated as that of some other component platforms (for example, (Reid et al. 2000)), however our observation was that the distinction was not clear or relevant to users. Removing it allowed more natural expression of system specifications and reduced user confusion.

Numerous component-based automotive systems are built on top of the AUTOSAR component platform (*AUTOSAR Release 4.2 Overview and Revision History* 2015). Like CAmkES, AUTOSAR is a static platform in which the precise system architecture is intended to be described at compile time. Rather than targeting a specific operating system, AUTOSAR targets an operating system specification called OSEK OS (*OSEK/VDX Operating System Specification, Version 2.2.3* 2005). OSEK OS is not expected to provide memory isolation, motivated in part by many of its target environments, Electronic Control Units (ECUs), lacking a Memory Management Unit (MMU). In contrast, CAmkES assumes mutual distrust between component instances and that the underlying operating system provides mechanisms to constrain their abilities. CAmkES is only intended to target hardware platforms with an MMU.

Beyond AUTOSAR, there are other less widely acknowledged component platforms that also target environments without a full MMU. For example, MMLite (Helander and Forin 1998), PECOS (Müller et al. 2001) and NesC (Gay et al. 2003). A common characteristic among these platforms is that they are aimed at addressing domain-specific challenges. For example, TinyOS (Levis et al. 2005), the primary consumer of NesC, is specifically intended for use in wireless sensor networks. In contrast, CAmkES aims to support a general purpose workload.

The terminology of component platforms is not always consistent. Several component platforms are framed as being toolkits for operating system construction. Notable examples are the previously mentioned MMLite, THINK (Fassino et al. 2002) and OSKit (Ford et al. 1997). While these may seem to have little in common with CAmkES and would relate more closely to a componentisation of seL4, this depends on where we draw the boundary of what is considered the "operating system." The kernel is only part of an operating system, and the operating system itself is a nebulous concept. Monolithic operating systems, in general, incorporate large amounts of functionality that is outside the purview of a microkernel-based operating system. Thus it is possible to see CAmkES as a toolkit for building a seL4-based operating system, and thereby sharing commonalities with these other platforms.

There are a number of component platforms that, like CAmkES, target L4-based microkernels, most prominently Genode (Feske and Helmuth 2007) and L4Re (Vogt, Döbel, and Lackorzynski 2010). There are commonalities in their implementations and Genode also has the ability to target seL4. However, the focus of neither is formal verification and both permit a level of architectural dynamism that is not supported by CAmkES. Verification of a dynamic component platform introduces further challenges that we do not attempt to deal with in the present work. As such, the use case of these other L4-based platforms somewhat differs from that of CAmkES.

[1] The CORBA specification, while detailed, is an informal document not amenable to verification. It provides sufficient information for the reader to build a conforming implementation, but does not yield a measure of confidence in the correctness of either the model or any implementation.

It is worth mentioning more general serialisation frameworks deployed in industry, the closest of which to CAmkES are Protocol Buffers (Varda 2008) and Cap'n Proto (Varda 2013). Although neither were used as a basis for CAmkES, there are similarities in their design and mechanisms. Observing the similarities in their functionality, this is not entirely coincidental as there is often an obvious, logical way of implementing a particular feature. As far as we are aware, neither provide any formal correctness guarantees.

3.2 Component Platform Correctness

The component platforms with which we compared CAmkES in the previous section for the most part have no correctness guarantees. While some involve detailed specifications, these specifications are fundamentally limited by being expressed in informal, imprecise language. In this section, we examine some of the concerted efforts to guarantee the correctness of component platforms.

One of the most widely deployed approaches in embedded systems is simply to rely on a minimal, manually auditable code base. Code minimality is often well aligned with, and also motivated by, other concerns such as the need for platform abstractions to collapse at compile time. This is observable in platforms such as NesC (Levis et al. 2005) and Pebble (Gabber et al. 1999), where many of the abstractions of the platform have no runtime overhead. However, we claim that relying on manual auditing alone for correctness can never provide sufficient correctness guarantees for a high-assurance component platform. Correctness by virtue of a minimal trusted computing base is a reasonable approach for a static code base, but a component platform's code is generated and differs from system to system. To gain confidence through manual auditing requires an *individual* inspection for each specific system. The validation effort for one system is not reusable on another system. While it could be argued that a more comprehensive alternative is to audit the template code of a component platform, we know of no reliable techniques for auditing such meta code. As a concrete example of the subtle ways this process can fail, Section 6.1 details a CAmkES bug that was not picked up by a manual audit of template code.

Pioneering work on the treatment of connectors as first class entities was achieved in the Wright framework (Allen and Garlan 1997). Wright shares similarities with the formalism we present in Chapter 5, though the authors do not go further to the level of code (as we do in Chapter 6) or draw a formal connection between their model and an implementation. Comparing Wright to our dynamic behavioural model alone, the two differ in several key ways. While both use CSP (Hoare 1985) as a basis, we provide a single choice operator while Wright offers both external and internal choice. The notions of "external" and "internal" do not have a well-defined meaning in our framework as the connectors and components are treated equivalently and neither are in opposition to an environment. Secondly Wright abstracts data in places that our formalism does not. For example, it is possible to express the contents of a shared memory region in our model, but not within Wright. However, this abstraction is not fundamental to Wright, rather the authors have taken this as a simplification. The specification of components within Wright is not described and the authors note this would require projection functions of the components' state. In fact, this is exactly the approach our work takes for representing components. Finally, verification of a component-based system in Wright generally involves showing conformance to an architectural style (Allen 1996). This is not an intended aim of ours, and we leave the correctness criteria open for the system designer to define.

There is a large amount of existing research in the area of correct-by-construction component-based systems. However, the majority of this work deals with the correctness of *components*, as opposed to the platform itself; a distinction introduced in Section 2.4. In particular, it is a common approach to solve architectural mismatch (Garlan, Allen, and Ockerbloom 1995) by deriving "interface compatibility" of a given system architecture. This line of work of seeks to show statically that an architecture represents a coherent pairing of component

instances; that is, component instances involved in a connection communicate in a compatible way. This has been attempted using state machines (Yellin and Strom 1994; Yellin and Strom 1997), Petri Nets (Silva, Gorgônio, and Perkusich 2008), constraint solving (Reid et al. 2000), attribute grammars (Batory and Geraci 1997) and numerous other approaches. We distinguish our work from such efforts both by the strength of the connections between our formalisms and the implementation of a component-based system, and by the expressiveness and robustness of the guarantees achievable in our work. The previous approaches we have referred to, orient a verification effort as targeting specific classes of properties. In contrast, we pursue full functional correctness as a generalised basis for any more specific property the system designer requires. Our approach can even serve as a basis for one of these previous, specialised techniques. The specialised technique can be used for reasoning about individual component instances, appealing to our formalisms for correctness of the platform itself.

Though not framed as component platform verification, there are similarities between the Ensemble system (Hayden 1998; Liu et al. 1999) and its associated Nuprl proof system and our work. The approach the authors use for translating code into their formalism (Kreitz 1997) shares aspects with our C-to-SIMPL translation, though their translation is from OCaml to Nuprl. However, similar to the work from the previous paragraph, the problem tackled by the authors is essentially interface compatibility. Arguably, in their domain of networking protocols, this is where the greatest benefit is to be had.

While interface compatibility is a necessary property for component system correctness, it is insufficient on its own. PECOS-CoCo (Genßler and Zeidler 2001; Genßler et al. 2002) attempts to take this further by statically validating a component architecture and additionally inserting runtime checks of functional properties. When comparing to our work, we show significantly stronger correctness properties and are able to achieve verification statically at compile time, obviating the need for runtime overhead. Not only does our work provide stronger guarantees, it precludes runtime failure which may be unacceptable in embedded environments that cannot be easily serviced.

In contrast to many other component verification efforts, the BIP component framework (Basu et al. 2011) aims to have a strong connection between a model of the behaviour of a component platform and its implementation. It is possible to generate code from a BIP model or derive a BIP model from an implementation (Basu et al. 2007). However, a significant gap still remains in the connection between code and model and the properties that can be proven on a BIP model are limited by its automated model checking approach. Like other existing work, BIP does not address the problem of guaranteeing *platform* correctness either.

There have been a few notable efforts towards correctness guarantees for L4-based component platforms. Nitpicker (Feske and Helmuth 2004; Feske and Helmuth 2005) provides a security-focused Graphical User Interface (GUI) environment, wherein colocated applications only have access to distinct regions of the display, with their I/O indirected through a server process. L4ReAnimator (Vogt, Döbel, and Lackorzynski 2010) monitors component execution, restarting crashed components, with the goal of increasing reliability and availability. To our knowledge, neither of these systems utilise formal methods. The correctness guarantees are derived from a minimal trusted computing base and the Principle of Least Authority (POLA). While this does achieve a measure of assurance, formal verification is able to produce a much stronger correctness guarantee than these techniques alone.

3.3 Decompositional Verification

The explicit boundaries between components and the isolation provided by the CAmkES platform partition the task of verifying a component-based system. By providing generated proofs of the platform glue code that are designed to integrate with hand written proofs, we are effectively providing natural points of decomposition for a larger proof. The manner in which the generated proofs and hand written proofs interact has similarities to, and is informed by, prior work on decompositional program verification.

The area of verification of concurrent programs has long been an active area of research. One of the most prominent techniques that has emerged from this field is Rely/Guarantee reasoning (Jones 1983; Owicki and Gries 1976). There are many superficial similarities between this work and ours (for example, Owicki and Gries' phrasing for preserving another process' precondition compared with our requirement to preserve the glue code's invariant, `inv`, as discussed in Chapter 6), albeit employed for differing purposes. However, ultimately we are solving a more constrained problem. User-provided code and glue code only share state in a very limited fashion; at any execution point, there is always a definitive "owner" of the shared state. In addition, the chained, sequential control flow ensures that the guarantees required by a fragment of code only need to hold on entry to and exit from that block. These are the primary reasons why we are able to achieve a greater level of automation than traditional Rely/Guarantee procedures.

Rely/Guarantee has previously been applied for reasoning about component-based systems by automatically generating assumptions and then using model checking or testing (Blundell, Giannakopoulou, and Păsăreanu 2006; Giannakopoulou, Păsăreanu, and Barringer 2002). While this gives some confidence in the correctness of a system, it can never exhaustively prove the absence of errors. In this respect, our approach has the advantage of an absolute guarantee; a correctness proof in any of the formalisms we introduce guarantees the implementation correct under the assumptions of the model.

A natural approach for decompositional program reasoning is separation logic (Reynolds 2002). The form of the Hoare triples in Chapter 6 suggest that it may be possible to rephrase these in separation logic with a suitable frame rule. Retrospectively, it has become apparent how to leverage separation logic for this formalism, but it was previously unclear that this would be a viable approach. Recent work has applied separation logic to proofs of seL4-based systems (Boyton 2014; Boyton et al. 2013), demonstrating that this is feasible. At this stage, reconstructing our formalism using separation logic would gain us little beyond more convenient syntax. However, it would provide a more natural environment for hand written proofs to build upon, and we consider it to be a worthwhile future improvement to our work.

3.4 Compiler Correctness

Generalising the functionality of a component platform's code generator, it is essentially a compiler. It accepts an input specification and produces executable code based on this specification. The design of component platform code generators typically share some common principles with conventional compilers. Thus the verification of a component platform can be compared to compiler verification efforts.

The most prominent low level compiler verification results have been the CompCert C compiler (Leroy 2006) and the Verisoft XT compiler (Böhme et al. 2010). While such work has commonality with ours in its goals (verification of a program-generating program) the strategies applied are mostly disjoint. The reason for this differentiation is that conventional compilers accept an executable input. That is, both the input and the output have executable semantics and it is the task of the compiler to preserve these semantics during its transformations. In the case of C, the standards define an abstract machine on which execution of the

source program takes place (ISO/IEC 2005). This direct equivalence does not exist in a component platform. While our work gives a well-defined semantics to the input to CAmkES (that we elaborate on in Chapter 4 and Chapter 7), this semantics does *not* correspond to the code generator's intended output in such a way that piecewise equivalence is a desirable property. With this in mind, our work draws on inspiration from compiler correctness research, but involves the application of different techniques.

Program synthesis is a technique for producing an executable program from an input that is not intended to be piecewise equivalent, an approach exemplified by Specware (Burstein et al. 2003; McDonald and Anton 2001). This can essentially be seen as the opposite to our approach in Chapter 6. Where we take code and generate a formal specification, Specware takes a formal specification and generates code. Both approaches have their strengths and weaknesses, but ours is particularly suitable for the domain of embedded systems where performance, code density and power consumption are key concerns. By allowing the glue code templates the full expressive power of the target language, it is possible to have much finer-grained control over the form of generated code. Related to program synthesis, is the synthesis of data structure representations, exemplified by systems like RelC (Hawkins et al. 2011) and Fiat (Delaware et al. 2015). However, these efforts are typically aimed at general programming language data structures and not directly applicable to the correctness of component platforms. We are not aware of any existing component platform with a formal specification that generates glue code from this specification and produces code-level correctness guarantees.

One specific technique for showing the correctness of a compiler-like transformation process is known as translation validation (Necula 2000; Pnueli, Siegel, and Singerman 1998). In this approach, instead of directly proving the correctness of the compiler, a correspondence between the input and output of the compiler is shown post hoc. If the proof of correct translation for a single instance is more straightforward than proving the compiler correct once and for all, this may be an attractive approach. Closely related is the idea of proof-carrying code (Necula 1997) wherein a program is accompanied by a proof of its correctness, obviating the need for the program consumer to trust it. The techniques we apply for component platform correctness are strongly aligned with these concepts. The CAmkES code generator is a large, complex program written in an interpreted language. Rather than attempt to prove the correctness of the generator and the runtime of its source language, we have designed it to produce accompanying proofs of the correctness of the artefacts it generates.

3.5 Summary

This chapter has detailed how our work relates to previous research in the areas of formal methods and component-based software engineering. In our discussion, we have given priority to work that we believe most closely relates to our methodology. Selected pieces of work in this chapter are referred to later in this thesis, when detailing specific technical similarities between our work and previous efforts. We have also provided context for where our work is situated and how it contributes to the state of the art in component platform verification.

Chapter 4

Static Architecture Model

In order to apply formal verification to a component platform, we need a formal abstract syntax and type system for its concepts. Component platforms are typically not accompanied by formal artefacts providing these features, and at the outset of our work the meaning of a CAmkES architectural description was only defined implicitly by the way in which the CAmkES code generator interpreted it. As a first step to the verification of CAmkES, it was necessary to formalise its syntactic concepts in such a way to allow reasoning about a given system within Isabelle/HOL.

This chapter discusses a formal model of the architecture of CAmkES systems. We present a set of static data type definitions. A specific component-based system can be unambiguously specified in terms of these definitions. Such a formal specification is derived entirely from the architectural description of a component system and is distinct from the component code itself. With reference to the distinction made in Section 2.4.3 and Section 2.4.4, this formalism aids a system designer in reasoning about the correctness of a system of *components*, not in gaining confidence in the component *platform* itself. The expressiveness of the formalism is limited, but it provides full automation. Any lemma that can be proven within this model can be discharged by automated proof tactics. This formalism has been the subject of a previous technical report (Fernandez, Klein, et al. 2013) and introduced in a previous paper (Fernandez, Kuz, et al. 2013). Sections of this chapter have been adapted from this prior work.

4.1 Utility

Section 2.2.1 described the types of component systems that can be constructed. Given the static architecture of CAmkES systems, there are certain properties that can be shown based on the architectural description alone. Informally, properties that are provable from the "boxes and lines" of a component-based system, without knowledge of what is inside the "boxes." To demonstrate this, consider the following CAmkES specification.

```
procedure Inf {
  /* Read a byte from remote memory */
  uint8_t read_byte(in unsigned int offset);

  /* Write a byte to remote memory */
  void write_byte(in unsigned int offset, in uint8_t value);
}

component Client {
  control;
  uses Inf inf;
}

component Server {
  provides Inf inf;
}

assembly {
  composition {
    component Client c1;
    component Client c2;
    component Server s1;
    component Server s2;

    connection seL4RPCSimple conn1(from c1.inf, to s1.inf);
    connection seL4RPCSimple conn2(from c2.inf, to s2.inf);
  }
}
```

The system can be depicted as in Figure 4.1. Note that there are two disconnected subsystems that make up this component system. Without knowing the details of the component code itself, it is possible to determine that `c1` and `s2` cannot communicate because there is no path between them. That is to say, there are no actions `c1` and `s2` could take in order to exchange information.[1] This is a property that is true of the *architecture*, rather than the *behaviour* of the system.

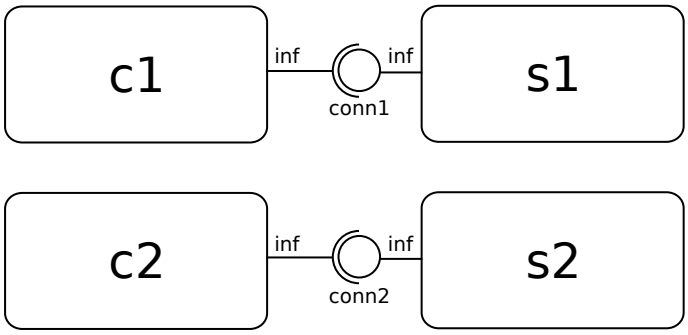

Figure 4.1: Example partitioned system

It may seem unusual for a component-based system to be architected this way, with certain component instances completely isolated from each other. However, this often occurs when two distinct systems have been consolidated to reduce hardware costs and increase utilisation. Though the cases where we can prove an information flow property within this formalism are limited, it has the advantage that all contained definitions are

[1] In this discussion we are excluding communication via side channels. Isolated subsystems in a CAmkES system may still be able to communicate via, for example, timing channels. Such covert communication is outside the scope of our present work, and addressed by other ongoing orthogonal efforts (Cock et al. 2014).

executable, and hence any correctness property that is expressible in this executable subset of HOL is provable automatically. To emphasise this point, a system designer can state a lemma in Isabelle/HOL capturing a property like the one described above and an automated tactic can discharge the lemma. This technique will only fail to discharge the lemma if it is false. We will return to this example and demonstrate such a proof in Section 4.5 after introducing the type system.

4.2 Type System

Recall that there are two inputs the system designer provides to define a CAmkES system: an architecture description and component source code. What we are representing in this abstraction is the former. There is a distinction in CAmkES between its ADL and IDL, with the former used to describe component types and system instantiation, and the latter used to define procedural interfaces. As discussed previously, this distinction is mostly arbitrary and considered a legacy aspect of CAmkES, but it remains explicit in this model. It serves some purpose here as these two types of symbols have differing constraints for their validity.

4.2.1 Symbols

CAmkES has two types of symbols that exist in separate namespaces. Symbols of the first type are used in IDL descriptions to identify names and parameters. These symbols are used during code generation and therefore need to be distinct within a specific interface.

```
1 type_synonym idl_symbol = string
```

Note that the `string` type in Isabelle/HOL is shorthand for a list of characters, `character list`.

The second type of symbols is used in ADL descriptions to identify components, connectors and other architecture-level entities. These symbols are also used during code generation, but are lexically scoped. For example, instantiated interfaces within separate components can have identical names.

```
1 type_synonym adl_symbol = string
```

Although both symbols map to the same underlying type, these have different constraints (for example, IDL symbols become direct substrings of code-level symbols and hence need to respect symbol naming restrictions for the target language(s)).

4.2.2 Methods

Methods are the elements that make up a CAmkES procedure, which will be described below. However, we begin with a treatment of their elements from a bottom up perspective.

Each method within a CAmkES procedure has a list of parameters defined by a type, direction and symbol name. Each method also has an optional return value type. Recall from Section 2.2.1.2 that the possible types of method parameters include high level, language-independent types and low level, language-dependent types. The low level types provide fixed width representations, floating point numbers and an opaque pointer type[2] that are specific to the C programming language. We define both the high level and low level types as the hierarchical Isabelle/HOL data type `primitive` as follows.

[2]The `uintptr_t` type serves the same role as the `uintptr_t` type in C, namely an integral representation of a pointer. It is intended to be used for transferring pointer-like data. The omission of a first class pointer type in CAmkES (that is, `void *`) is not accidental.

```
datatype number =
    -- "High level types"
    UnsignedInteger
  | Integer
  | Real
  | Boolean
    -- "C-specific types"
  | uint8_t
  | uint16_t
  | uint32_t
  | uint64_t
  | int8_t
  | int16_t
  | int32_t
  | int64_t
  | double
  | float
  | uintptr_t

datatype textual =
    -- "High level types"
    Character
  | String
    -- "C-specific types"
  | char

datatype primitive =
    Numerical number
  | Textual textual
```

Arrays with a fixed element type are supported as method parameters in CAmkES. Similar to primitive types, using a high level type for the elementary type of an array allows it to be implemented or used in any component, while using a low level type prevents implementing or using it in a component not written in C. Arrays of arrays and multidimensional arrays are not supported.[3] All arrays are inherently variable length. In C, this is implemented in generated code with an extra function parameter for array length.

Thus the type of a method parameter is either a primitive or an array, wherein both have a fundamental primitive type.

```
datatype param_type =
    Primitive primitive
  | Array primitive
```

For the purposes of this static architectural model, we do not need to represent the implementation details of the individual types. That is, it is not necessary or relevant to know, for example, that the `uint8_t` type is a single unsigned byte. Similarly, the serialised data representation of each type is left unspecified in this model.

Interfaces define a communication payload between component instances that are typically hosted in separate virtual address spaces. In this environment, a transmitted pointer cannot be sensibly dereferenced as the address it contains points to different data in the sender's and receiver's address spaces. To avoid confusion, we omit a pointer type. In the context of shared memory where component instances may legitimately wish to exchange pointers, CAmkES provides an alternative mechanism for accomplishing this. More information is available in the CAmkES documentation (Trustworthy Systems Team 2015).

[3] Note that hierarchical or multidimensional arrays can be straightforwardly mapped onto a one dimensional array by the user if needed.

Rather than having a single return value per procedure method, each method parameter can be an input parameter, an output parameter, or both.[4]

```
datatype param_direction =
    InParameter (* also covers 'refin' *)
  | OutParameter
  | InOutParameter
```

Each parameter has a type and direction, as described, and an identifier.

```
record parameter =
  p_type        :: param_type
  p_direction   :: param_direction
  p_name        :: idl_symbol
```

Methods are a list of parameters, an optional return type and an identifier.

```
record "method" =
  m_return_type :: "param_type option"
  m_name        :: idl_symbol
  m_parameters  :: "parameter list"
```

The translation from procedure methods in CAmkES' textual input format to their representation in Isabelle/HOL is straightforward. Consider the following CAmkES method.

```
int foo(in string s);
```

This is translated into the following Isabelle/HOL representation.

```
  "⦇m_return_type = Some (Primitive (Numerical Integer)),
    m_name = ''foo'',
    m_parameters = [
      ⦇p_type = Primitive (Textual String),
       p_direction = InParameter,
       p_name = ''s''⦈
    ]⦈"
```

The translated representation appears verbose, but is composed of simple data types and precisely preserves the information of the originating declaration. The translation process is entirely mechanical, and CAmkES automates this for the system designer.

The level of abstraction of this representation is similar to a textual CAmkES architecture description. That is, little is known about the implementation of interface methods. This abstraction facilitates portability of component system descriptions between source code languages and operating environments.

[4]The CAmkES grammar, in addition to `in`, `out`, and `inout` parameters, supports a fourth direction, `refin`. This is a C-specific optimisation and `refin` parameters are semantically equivalent to `in` parameters. We do not differentiate them explicitly in this model, as they can be adequately represented using the `InParameter` type.

4.2.3 Interfaces

Connections between two components are made from one interface to another. This model distinguishes between the three types of communication abstraction presented by CAmkES. We have already seen the form of procedure types. Event types are represented as natural numbers (recall that events carry no data other than their type) and dataport types as an optional string identifying a target language type. If a dataport type is `None`, we consider it to be a built in type named `Buf`.

```
type_synonym procedure = "method list"
type_synonym event     = nat
type_synonym dataport  = "string option"

datatype interface =
    Procedure procedure
  | Event event
  | Dataport dataport
```

Note that connectors (the type of connections) are expected to link two interfaces of identical type.

4.2.4 Connectors

Two component instances are connected via a connection. A connection is an instantiation of a particular *connector*. The type of a connector is an abstraction of the underlying communication mechanism. Connectors can be categorised into two distinct types: native connectors and hardware connectors.[5]

```
datatype connector_type =
    Native native_connector_type
  | Hardware hardware_connector_type
```

Native connectors map directly to implementation mechanisms. These are purely a software mechanism and are the types of connectors that are found in almost all component platforms.

```
datatype native_connector_type =
    AsynchronousEvent -- "an asynchronous notification"
  | RPC -- "a synchronous channel"
  | SharedData -- "a shared memory region"
```

Recalling from Section 2.2.1 that hardware devices are modelled as components in CAmkES, hardware connectors are used to connect the interface of a device to the interface of a software component. Devices must be connected using the connector type that corresponds to the mode of interaction with the device.[6]

```
datatype hardware_connector_type =
    HardwareMMIO -- "memory mapped IO"
  | HardwareInterrupt -- "device interrupts"
  | HardwareIOPort -- "x86 I/O ports"
```

Connectors are distinguished by the mode of interaction they enable. The reason for this will become clearer in the following sections.

[5] As discussed previously, we do not support CAmkES' hardware connectors, but they appear in this model for completeness. By being able to represent hardware connectors in this formalism we ease future extensions in this direction.

[6] Note that we represent x86 I/O ports in this data type even though we are targeting ARM. Currently this member is never used, but this has been included in the data type to ease future extensions.

```
datatype connector =
    SyncConnector connector_type
  | AsyncConnector connector_type
  | MemoryConnector connector_type
```

4.2.5 Components

Functional entities in a CAmkES system are represented as components. These are reusable collections of source code with explicit descriptions of the exposed methods of interaction ("interfaces" described previously).

Recall from Section 2.2.1 that components have three distinct modes of communication:

1. Synchronous communication over procedures (RPC). This communication is analogous to a function call and happens over a channel established from a `requires`[7] interface to a `provides` interface.

2. Asynchronous communication using events. This is suitable for things like signalling and happens over a channel from an `emits` interface to a `consumes` interface.

3. Bidirectional communication via shared memory. This is suitable for passing large data between components. It happens over a channel between two `dataports`.

We represent a component type as an Isabelle/HOL record with a member for each category of interface the component may have. We also represent whether the component has an active thread of control with a dedicated field.

```
record component =
  control     :: bool
  requires    :: "(adl_symbol × procedure) list"
  provides    :: "(adl_symbol × procedure) list"
  dataports   :: "(adl_symbol × dataport) list"
  emits       :: "(adl_symbol × event) list"
  consumes    :: "(adl_symbol × event) list"
  attributes  :: "(adl_symbol × param_type) list"
```

Note that a component's attributes are represented in the above record. We represent this information in this model purely to support a faithful information-preserving translation. The values of attributes are not examined or constrained within this formalism.

4.2.6 Assembling a System

A complete system is formed by instantiating component types that have been defined, interconnecting these instances and specifying a system configuration. Connections are specified by the two interfaces they connect and the communication mechanism in use.

```
record connection =
  conn_type   :: connector
  conn_from   :: "(adl_symbol × adl_symbol)"
  conn_to     :: "(adl_symbol × adl_symbol)"
```

[7] A `requires` interface is entirely equivalent to a `uses` interface in the CAmkES grammar. The reason for differing terminology is purely pragmatic; "uses" is a reserved keyword in Isabelle/HOL. It was decided not to change the CAmkES keyword as this would unnecessarily confuse established developers accustomed to `uses`.

In the preceding definition, each of the interfaces is identified by a pair of symbols corresponding to a component instance and its interface which participate in the connection. So, for example, if a connection is from `foo.bar` (the `bar` interface of component instance `foo`), its corresponding Isabelle/HOL definition will have a `conn_from` field of `(''foo'', ''bar'')`.

A composition block is used to contain all component instances of the system and the connections that define their communication with each other.

```
record composition =
  components  :: "(adl_symbol × component) list"
  connections :: "(adl_symbol × connection) list"
```

Configurations are used as a way of adding extra information to a component or specialising the component in a particular context. The attributes available to set are specified in the definition of the component[8], as indicated above. These attributes are accessible to the component code.

```
type_synonym configuration = "(adl_symbol × adl_symbol × string) list"
```

Finally the system specification is expressed at the top level as an assembly. This extra level of abstraction allows more flexible reuse of compositions and configurations.

```
record assembly =
  composition   :: "composition"
  configuration :: "configuration option"
```

4.3 Wellformed Definitions

On specifications within the type system just described, it is possible to define a generic property of valid specifications. While this property does not guarantee a system conforms to the designer's expectations, it does guarantee internal consistency of the specification itself. We call a static architecture model "wellformed" if it has this internal consistency.

This section provides a definition of what it means to be wellformed for each syntactic element that captures the necessary conditions for it to represent a valid system architecture. Any wellformed system is capable of being instantiated in a way that corresponds to its ADL.

4.3.1 Wellformed Interfaces

A procedure method is considered wellformed if the symbols of its name and parameters are distinct. This constraint ensures the code generation process will produce valid code in the target language.[9]

[8] This discussion excludes infrastructure attributes that are implicit and used for configuring runtime characteristics of the platform (for example, setting thread priority). The infrastructure attributes do not have relevance in this model as runtime behavioural aspects of the system are not represented. More information about infrastructure attributes is provided in the CAmkES reference manual for the curious reader (Trustworthy Systems Team 2015).

[9] Depending on the target language, there may be additional conditions on the validity of CAmkES symbols that have an effect on symbols in generated code. For example, C identifiers must start with an alphabetic character or underscore and must not collide with certain keywords. Rather than represent these details, we omit them from our definitions, leaving the notion of a wellformed interface language-independent. It should be clear how to trivially extend these definitions to capture specific rules of a target language's syntax.

```
definition
  wellformed_method :: "method ⇒ bool"
where
  "wellformed_method m ≡
    (m_name m ∉ set (map p_name (m_parameters m)) ∧
     distinct (map p_name (m_parameters m)))"
```

The code generated for a procedure is within a single namespace and thus the names of all methods in a procedure must be distinct. Note that we also require procedures to have at least one method. Procedures without methods have no use, and thus this serves as a sanity check for users.

```
definition
  wellformed_procedure :: "procedure ⇒ bool"
where
  "wellformed_procedure i ≡
    (i ≠ []) ∧
    (∀x ∈ set i. wellformed_method x) ∧
     distinct (map m_name i)"
```

Recall from Section 2.2.1 that events carry no data. The only property an event type has is its distinctness from other event types. Thus we represent event types as distinct natural numbers. These identifiers are required to be less than 32 due to implementation-specific concerns.[10] This is not fundamental to the design of this model and this constraint could be removed without excessive effort.

```
definition
  wellformed_event :: "event ⇒ bool"
where
  "wellformed_event e ≡ e < 32"
```

Dataports do not have any attributes beyond their type, so they are trivially wellformed.

```
definition
  wellformed_dataport :: "dataport ⇒ bool"
where
  "wellformed_dataport d ≡ True"
```

The conditions for an interface to be wellformed can now be defined hierarchically.

```
fun
  wellformed_interface :: "interface ⇒ bool"
where
  "wellformed_interface (Procedure p) = wellformed_procedure p"
 |"wellformed_interface (Event e) =  wellformed_event e"
 |"wellformed_interface (Dataport d) = wellformed_dataport d"
```

4.3.2 Wellformed Components

For a component to be wellformed, we require that its interfaces themselves are wellformed. To instantiate a component successfully we also need to ensure that the names of the interfaces are distinct. References to a component's interfaces are by name, hence we need to ensure that such references are unambiguous. These two requirements are captured in the following definition.

[10] This limitation is motivated by the way in which event identifiers are multiplexed onto asynchronous endpoint fields in seL4. Asynchronous endpoints have a single word of data which is used by CAmkES as a bit set of which events have been received. This design necessitates a maximum of 32 incoming events (identifiers 0 to 31). In practice, users have not found this to be restrictive in the construction of real systems, but the design could easily be restructured to remove this limitation.

```
definition
  wellformed_component :: "component ⇒ bool"
where
  "wellformed_component c ≡

     (* No symbol collisions *)
    (distinct (map fst (requires c) @ map fst (provides c) @
     map fst (dataports c) @ map fst (emits c) @ map fst (consumes c)) ∧

     (* No C symbol collisions *)
    (∀x ∈ set (requires c). wellformed_procedure (snd x)) ∧
    (∀x ∈ set (provides c). wellformed_procedure (snd x)) ∧

     (* Events valid *)
    (∀x ∈ set (emits c). wellformed_event (snd x)) ∧
    (∀x ∈ set (consumes c). wellformed_event (snd x)) ∧

     (* Dataports valid *)
    (∀x ∈ set (dataports c). wellformed_dataport (snd x)))"
```

4.3.3 Wellformed Connectors

A wellformed connector primarily requires internal consistency of its mechanism. That is, that the type of communication it performs is coherent with its mode of interaction.

```
definition
  wellformed_connector :: "connector ⇒ bool"
where
  "wellformed_connector c ≡ case c of
       SyncConnector t ⇒ case t of
           Native n ⇒ n ∈ {RPC}
         | Hardware h ⇒ h ∈ {HardwareIOPort}
     | AsyncConnector t ⇒ case t of
           Native n ⇒ n ∈ {AsynchronousEvent}
         | Hardware h ⇒ h ∈ {HardwareInterrupt}
     | MemoryConnector t ⇒ case t of
           Native n ⇒ n ∈ {SharedData}
         | Hardware h ⇒ h ∈ {HardwareMMIO}"
```

Note that this particular constraint could have been encoded directly in the type system, but we chose to express this in the wellformed predicate for flexibility during development. In practice it has not caused problems, and either formulation would have been suitable.

4.3.4 Wellformed Connections

The only internal consistency property for a connection (the instantiation of a connector) is that its references are valid symbols. This is captured at a higher level in the definition of a wellformed component system composition. For this reason, we define a trivial predicate for wellformed connections.

```
definition
  wellformed_connection :: "connection ⇒ bool"
where
  "wellformed_connection c ≡ True"
```

4.3.5 Symbol Resolution

A CAmkES specification uses textual identifiers to indicate references to defined entities. For example, in the following specification, `Foo` on line 6 references the procedure defined in lines 1-3.

```
1 procedure Foo {
2   void method(void);
3 }
4
5 component Bar {
6   uses Foo f;
7 }
```

The resolution of these symbolic textual references is represented formally in this model. For each outgoing interface (`requires` and `emits`), we require that there is a single connection with a matching component instance.[11] Here we use the non-standard binder, $\exists 1$, for "there exists exactly one."

```
1  definition
2    refs_valid_interfaces :: "adl_symbol ⇒ (adl_symbol × procedure) list ⇒
3      (adl_symbol × event) list ⇒ (adl_symbol × connection) list ⇒ bool"
4  where
5    "refs_valid_interfaces component_instance procedures events conns ≡
6
7      (* Each outgoing procedure is connected *)
8      (∀x ∈ set procedures.
9       (∃1 y ∈ conns. from_component (snd y) = component_instance ∧
10                    from_interface (snd y) = fst x)) ∧
11
12     (* Each outgoing event is connected *)
13     (∀x ∈ set events.
14      (∃1 y ∈ conns. from_component (snd y) = component_instance ∧
15                    from_interface (snd y) = fst x))"
```

Incoming interfaces can be left disconnected in a system with no adverse effects. Thus, the only property we need of components is the correctness of their `requires` and `emits` interfaces.

```
1  definition
2    refs_valid_components ::
3      "(adl_symbol × component) list ⇒ (adl_symbol × connection) list ⇒ bool"
4  where
5    "refs_valid_components comps conns ≡ ∀x ∈ set comps.
6      refs_valid_interfaces (fst x) (requires (snd x)) (emits (snd x)) conns"
```

[11] Recall from Section 2.2.1 that CAmkES supports connections that have multiple component instances at either end. However, for the purposes of this work, we deal only with one-to-one connections.

Each connection must be connecting interfaces of the same underlying type.

```
definition
  refs_valid_connection :: "connection ⇒ (adl_symbol × component) list ⇒ bool"
where
  "refs_valid_connection x component_list ≡

    wellformed_connector (conn_type x) ∧
    (case conn_type x of

        SyncConnector _ ⇒
            (* Corresponding procedures exist *)
          (∃1 y ∈ component_list. to_component x = fst y ∧
                                  does_provide (snd y) (to_interface x)) ∧
          (∃1 y ∈ component_list. from_component x = fst y ∧
                                  does_require (snd y) (from_interface x))

      | AsyncConnector _ ⇒
            (* Corresponding events exist *)
          (∃1 y ∈ component_list. to_component x = fst y ∧
                                  does_consume (snd y) (to_interface x)) ∧
          (∃1 y ∈ component_list. from_component x = fst y ∧
                                  does_emit (snd y) (from_interface x))

      | MemoryConnector _ ⇒
            (* Corresponding dataports exist *)
          (∃1 y ∈ component_list. to_component x = fst y ∧
                                  has_dataport (snd y) (to_interface x)) ∧
          (∃1 y ∈ component_list. from_component x = fst y ∧
                                  has_dataport (snd y) (from_interface x)))"
```

```
definition
  refs_valid_connections ::
    "(adl_symbol × connection) list ⇒ (adl_symbol × component) list ⇒ bool"
where
  "refs_valid_connections conns comps ≡
    ∀x ∈ set conns. refs_valid_connection (snd x) comps"
```

Putting these together, we have that the validity of references in a composition is the validity of references in its contained component instances and connections.

```
definition
  refs_valid_composition :: "composition ⇒ bool"
where
  "refs_valid_composition c ≡
    refs_valid_components (components c) (connections c) ∧
    refs_valid_connections (connections c) (components c)"
```

4.3.6 Overall System

We obtain a guarantee of the correctness of a component composition by composing the required properties of its constituents.

```
definition
  wellformed_composition :: "composition ⇒ bool"
where
  "wellformed_composition c ≡

     (* This system contains ≥ 1 active component *)
    (∃x ∈ set (components c). control (snd x)) ∧

     (* All references resolve *)
    refs_valid_composition c ∧

     (* No namespace collisions *)
    distinct (map fst (components c) @ map fst (connections c)) ∧

     (* All components are valid *)
    (∀x ∈ set (components c). wellformed_component (snd x)) ∧

     (* All connections are valid *)
    (∀x ∈ set (connections c). wellformed_connection (snd x))"
```

The correctness of a configuration is trivial, as the type system already constrains it to only have valid values.

```
definition
  wellformed_configuration :: "configuration ⇒ bool"
where
  "wellformed_configuration conf ≡ True"
```

Thus we have the correctness of an assembly as simply the correctness of its members.

```
definition
  wellformed_assembly :: "assembly ⇒ bool"
where
  "wellformed_assembly a ≡
    wellformed_composition (composition a) ∧ (case (configuration a) of
      None ⇒ True
    | Some x ⇒ wellformed_configuration x)"
```

These definitions are sufficient to formally define what it means for CAmkES specifications to be wellformed. We require a specification to be wellformed to instantiate it as a running system. That is, if the CAmkES code generator is supplied with a specification that is not wellformed, we guarantee nothing about its output and permit it to generate incorrect code.[12] The system designer is expected to defend themselves against this scenario by showing their specification is wellformed.

It can be observed that all the definitions we have provided in this chapter are executable. This naturally leads to them being automatically decidable on a concrete specification.[13] Thus, the obligation on the system designer to show their specification to be wellformed can be discharged entirely automatically by Isabelle/HOL.

[12] This may seem unusual, but is a relatively common style of specifying compiler correctness. That is, the compiler, when fed invalid input, is permitted to do anything including produce the infamous nasal demons (Woods 1992).

[13] A side effect of an executable model is that we could also use code extraction to obtain a verified wellformed checker for CAmkES specifications. This was not required for the current work and has not been attempted, but would be a trivial extension to the existing implementation.

4.4 Example System

To demonstrate the utility of the model introduced in this chapter, we return to the example system from Section 4.1. Its specification can be automatically translated into Isabelle/HOL terms using the definitions we have introduced in this chapter. The procedure from the specification is translated into the following formal representation.

```
definition
  Inf :: procedure
where
  "Inf ≡ [⦇ m_return_type = Some (Primitive (Numerical uint8_t)),
            m_name = ''read_byte'',
            m_parameters = [⦇ p_type = Primitive (Numerical UnsignedInteger),
                              p_direction = InParameter,
                              p_name = ''offset'' ⦈] ⦈,
          ⦇ m_return_type = None,
            m_name = ''write_byte'',
            m_parameters = [⦇ p_type = Primitive (Numerical UnsignedInteger),
                              p_direction = InParameter,
                              p_name = ''offset'' ⦈,
                           ⦇ p_type = Primitive (Numerical uint8_t),
                              p_direction = InParameter,
                              p_name = ''value'' ⦈] ⦈]"
```

Observe that the Isabelle/HOL definition closely matches the input specification in the level of detail and information encapsulated.

The `Client` component has a corresponding definition containing its single interface and showing that it has a thread of control.

```
definition
  Client :: component
where
  "Client ≡ ⦇control = True,
             requires = [(''inf'', Inf)],
             provides = [],
             dataports = [],
             emits = [],
             consumes = [],
             attributes = []⦈"
```

The `Server` component's definition appears similar, though note that it has no thread of control.

```
definition
  Server :: component
where
  "Server ≡ ⦇control = False,
             requires = [],
             provides = [(''inf'', Inf)],
             dataports = [],
             emits = [],
             consumes = [],
             attributes = []⦈"
```

These are the only generated component definitions. Although there are four component *instances* in this system, there are only two component *types*. Each definition is instantiated twice, once per component instance.

```
definition
  c1 :: component
where
  "c1 ≡ Client"

definition
  c2 :: component
where
  "c2 ≡ Client"

definition
  s1 :: component
where
  "s1 ≡ Server"

definition
  s2 :: component
where
  "s2 ≡ Server"
```

The connections are given as definitions that capture their type (a connector) and each of their ends as textual references.

```
definition
  conn1 :: connection
where
  "conn1 ≡ ⦇conn_type = seL4RPCSimple,
            conn_from = (''c1'', ''inf''),
            conn_to = (''s1'', ''inf'')⦈"

definition
  conn2 :: connection
where
  "conn2 ≡ ⦇conn_type = seL4RPCSimple,
            conn_from = (''c2'', ''inf''),
            conn_to = (''s2'', ''inf'')⦈"
```

Finally these definitions are combined in a composition, configuration and ultimate assembly.

```
definition
  composition' :: composition
where
  "composition' ≡ ⦇
    components = [(''c1'', c1), (''c2'', c2), (''s1'', s1), (''s2'', s2)],
    connections = [(''conn1'', conn1), (''conn2'', conn2)]⦈"

definition
  configuration' :: "configuration option"
where
  "configuration' ≡ None"

definition
  assembly' :: assembly
where
  "assembly' ≡ ⦇composition = composition',
                configuration = configuration'⦈"
```

Each term has an accompanying proof that it is wellformed. We show the proof corresponding to the assembly as an example.

```
lemma wf_assembly': "wellformed_assembly assembly'"
  by code_simp
```

To clarify the definitions provided in this section, all Isabelle/HOL artefacts are produced automatically from the designer's initial textual specification. That is, the translation of the specification into Isabelle/HOL and the proof that it is wellformed are fully automatic. The fact that the proof can be completed automatically is not due to the simplicity of this example, but rather the executability of this model, as discussed in the previous section. Any given CAmkES specification can be proven wellformed automatically in this manner. There is no ambiguity in the model and thus conversely, a failure of this proof definitively indicates that a specification is *not* wellformed.

4.5 Information Flow

We move now to demonstrating how simple information flow properties can be automatically proven in this formalism. We can phrase a definition that extracts the connections in the system and specifies a set of connected component instance names.

```
definition
  links :: "(string × string) set"
where
  "links ≡ set (map (λc. (fst (conn_from c), fst (conn_to c)))
                (map snd (connections (composition assembly'))))"
```

Observe that this set describes a relation between connected component instances. From this, we can write a lemma in terms of the transitive closure of this relation that states `c1` is unreachable from `s2`. As with proving the specification is wellformed, this property is discharged automatically.

```
lemma "{(''c1'', ''s2''), (''s2'', ''c1'')} ∩ links⁺ = {}"
  by code_simp
```

Any property in the executable subset of Isabelle/HOL can be proven automatically in this manner. For example, on a given system we could prove properties such as "every connection to a network interface passes through an encryption component" or "all input from untrusted components passes through a sanitising component." More powerfully, such properties can be written as executable predicates on an architecture specification (`assembly ⇒ bool`) and Isabelle/HOL's code extraction can be used to generate an independent program to test this property of arbitrary future specifications.

4.6 Limitations

The properties that can be expressed and proven on this model are limited. For instance, it is generally not possible to prove properties about directly connected component instances. However, the advantage of this model is its executability. A property in the expressive subset of this model can be proven automatically. This makes it particularly attractive for system designers who are not formal methods experts. The only complexity in reasoning is constructing the property to be proved.

A natural consequence of the limited expressiveness of this model is that a property that cannot be proven is not inherently false. There are certain properties that depend on both the system architecture and behaviour of individual component instances. For example, even if `c1` and `s2` were transitively connected in the previous example, it may still be true that there is no information flow between them because they never use the connections available to them. Such properties cannot be proven on this model, and are the motivation for the more expressive, lower level models discussed in Chapter 5 and Chapter 6.

4.7 Summary

In this chapter, we have shown a formalism for describing the static architecture of component systems. Definitions for a specific component system are automatically generated from the designer's architecture specification. The formalism is comparatively basic, but specification validation and the proof of simple properties can be performed automatically. We have demonstrated how this can be applied in the context of an example system.

The pervasive automation within this formalism aids its use by non-experts. A designer who understands their architecture and its intended separation properties can check that it is wellformed and then prove isolation lemmas with little introduction to Isabelle/HOL or formal verification. In part, the value of this model is in lowering the barrier to the use of formal methods in CBSE.

Chapter 5

Dynamic Behavioural Model

The behaviour of a software system at runtime is dictated by its code. A component-based system contains code the designer has provided for its components and code introduced by the component platform. Understanding and reasoning about complex systems requires a model of what this collection of code does. As with many software systems, there is an upper bound on what is provable statically on a representation of a component architecture, and certain properties require analysis of the dynamic behaviour[1] of the contents of the system.

The topic of this chapter is a model of the dynamic behaviour of CAmkES systems. This model is built on a message passing formalism for imperative programs and allows the expression of an upper bound on the runtime behaviour of a component system. As with the static architecture model from the preceding chapter, generated artefacts for a specific component system in this model are based entirely on the system's architecture specification, and are independent of the component code provided. We infer a component instance's behaviour from the constraints imposed by the architecture. This formalism provides greater expressiveness than the static architecture model, but at the cost of full automation. Proofs in this formalism will typically require some manual reasoning, though we automate the production of system artefacts that do not affect correctness. This material has been the subject of a previous technical report (Fernandez, Gammie, et al. 2013) and part of a previous paper (Fernandez, Kuz, et al. 2013) and some content from this chapter is adapted from these.

We begin by describing what is possible to express within this model and its utility to a system designer. We then introduce the foundations of the formalisation and show example generated theories for each CAmkES communication mode. Finally, we return to the initial example to demonstrate how to prove properties that rely on runtime behaviour.

5.1 Expressiveness

Chapter 4 described a formalism suitable for proving properties that are *statically* true of the architecture of a component-based system. That is, properties that are provable without needing to consider the behaviour of component instances. This chapter is concerned with a formalism in which properties that rely on the runtime behaviour of component instances can be specified. Consider a scenario involving component instances that

[1] Within this chapter, we use the phrase "dynamic behaviour" to refer to the system's execution, in opposition to its "static architecture." We spell this out to avoid the misinterpretation that we are instrumenting the system at runtime. All our formal analysis, including the work in this chapter, is performed offline.

are transitively reachable from one another. The following system contains three component instances that, if we view connections as bidirectional edges, all lie within the same strongly connected group.

```
procedure Lookup {
  int get_value(in int key);
}

component Client {
  control;
  uses Lookup l;
}

component Filter {
  provides Lookup external;
  uses Lookup backing;
}

component Store {
  provides Lookup l;
}

assembly {
  composition {
    component Client client;
    component Filter filter;
    component Store store;

    connection seL4RPCSimple one(from client.l, to filter.external);
    connection seL4RPCSimple two(from filter.backing, to store.l);
  }
}
```

Figure 5.1: Example interconnected system

This system, depicted in Figure 5.1, has a dictionary of integers hosted in `store` that can be looked up through the provided interface. The instance `filter` interposes on `client`'s access to this dictionary in order to restrict access to the keys it can look up. Consider the source code of the component type `Store` to be the following.

```
int l_get_value(int key) {
  if (key == 0) {
    /* Secret value */
    return 42;
  } else if (key == 1) {
    return 7;
  } else {
    return -1;
  }
}
```

And the following as the source code for the component type `Filter`.

```
int external_get_value(int key) {
  int value = backing_get_value(key);
  if (value == 7) {
    return value;
  } else {
    /* Deny access to secrets */
    return -1;
  }
}
```

Observe that `Filter` never returns the secret value 42. In our example system, the only way `client` can receive values from `store` is via `filter`, so we know `client` cannot receive 42. However, this is not provable within the formalism of Chapter 4. What is necessary but lacking, is a representation of the behaviour of `filter` at runtime. The following sections outline a formalism that allows this specification. We will return to this example in Section 5.10 to demonstrate how to accomplish this after detailing the model's foundations.

5.2 CIMP

We build on top of a previously existing language called CIMP (Gammie 2015). CIMP is a small concurrent imperative language with synchronous message passing, in the style of communicating sequential processes (Hoare 1985). The sequential part of the language is a standard, minimal, Turing-complete While-language. Though the language was not initially designed with CAmkES in mind, it is sufficient to express the semantics of CAmkES glue code and the behaviour of small, trusted component instances.

The message passing mechanism is a slight variation of standard synchronous message passing instructions, `send` and `receive`, that would map directly to seL4's synchronous IPC. The standard mechanism in labelled transition systems would identify the message with a message label and potentially a payload. In our setting, we extend this concept to the instructions `Request` and `Response`, that come with two labels, one for a question and one for the corresponding answer that is provided in the same execution step. The standard mechanism can be obtained simply by leaving out answers, that is by setting the answer type to `unit`.

We additionally allow these messages to depend on the state when they are sent as a question and to modify the local state to store the content of an answer. This variation allows us to conveniently use the same mechanism for modelling memory, where the response from memory is instantaneous, or to model asynchronous messages, where the effect is simply to store the message in a buffer. We will return to this possibility in Section 5.5.

Below follows the data type for sequential commands in the language. We first define the type of (shallowly embedded) boolean expressions to be a function from the state `'s` to `bool`.

```
type_synonym 's bexp = "'s ⇒ bool"
```

In this data type, and in many of those that follow, the definition is parameterised with a type for the state. This gives the system designer the flexibility to represent only the elements of the system state that are of interest to them. This is crucial for supporting a model that is expressive enough to capture desirable correctness properties, but sufficiently constrained to scale to large systems.

The type of sequential commands itself is parameterised by a type `'a` for answers, a type `'q` for questions and a type `'s` for the state of the program. The alternatives of the data type are standard:

- Skip, which does nothing;
- A local operation that can model any function on the local state `'s`, such as a variable assignment;
- Sequential composition;
- If-then-else;
- While loops with a boolean expression and a body;
- Binary non-deterministic choice;
- Message sending (`Request`); and
- Message receiving (`Response`).

In Isabelle/HOL, this appears as follows. There are also convenient notational alternatives provided for some members. For example, we can express a while loop more naturally as `WHILE condition DO body`. Note that this data type is recursive; for example a `Seq` contains two other `com` terms.

```
datatype ('a, 'q, 's) com
  = Skip  ("SKIP")
  | LocalOp "'s ⇒ 's set"
  | Seq "('a,  'q, 's) com" "('a,  'q, 's) com"
      (infixr ";;"  60)
  | If "'s bexp" "('a,  'q, 's) com" "('a,  'q, 's) com"
      ("(IF _/ THEN _/ ELSE _)"  [0, 61] 61)
  | While "'s bexp" "('a,  'q, 's) com"
      ("(WHILE _/ DO _)"  [0, 61] 61)
  | Choose "('a,  'q, 's) com" "('a,  'q, 's) com"
      (infixl "⊔" 20)
  | Request "'s ⇒ 'q set" "'a ⇒ 's ⇒ 's set"
  | Response "'q ⇒ 's ⇒ ('s × 'a) set"
```

For convenience, we introduce infinite loops as an abbreviation. They are, for instance, used in event handling loops.

```
abbreviation
  LOOP_syn ("LOOP/ _")
where
  "LOOP c ≡ WHILE (λ_. True) DO c"
```

After the sequential part, we are now ready to define the externally-visible communication behaviour of a process. Processes execute by taking labelled steps, $\texttt{p} \rightarrow_{\texttt{label}} \texttt{p'}$. A process can make three kinds of labelled steps: internal τ steps, message sends, and message receives. Both of the latter are annotated with the action/payload of both the request and instantaneous response (if any) of that message. Again, this data type is parameterised by two types and offers notational alternatives for each member.

```
datatype ('a, 'q) seq_label
  = SL_Internal ("τ")
  | SL_Send 'q 'a ("≪_, _≫")
  | SL_Receive 'q 'a ("≫_, _≪")
```

The following inductive definition now gives the small-step or structural operational semantics of the sequential part of the language. The semantics judgement is a relation between configurations, labels, and follow-on configurations. A configuration consists, as is usual in such settings, of a command and local state `'s`.

The two interesting rules are at the beginning: a `Request action val` command can make a step labelled as $\ll\alpha,\ \beta\gg$ from state `s` to `s'` if α is one of the actions that is enabled by `action` in state `s`, and if `val` extracts `s'` from the response β in `s`. Similarly, a `Response action` command progresses from `s` to `s'` with label $\gg\alpha,\ \beta\ll$ if β is among the possible responses for the request α, and if `s'` is in the possible successor states after potentially extracting α's payload into the local state.

The other rules are a standard small-step semantics for a minimal non-deterministic imperative language. Local and terminating steps produce τ transitions and all other labels are passed through appropriately. In preference to showing the raw Isabelle/HOL definition, we present its type signature and then the rule alternatives in the style of proof trees for clarity.

```
inductive small_step ::
  "('a,  'q, 's) com × 's ⇒ ('a, 'q) seq_label ⇒
   ('a,  'q, 's) com × 's ⇒ bool"
```

Request: $$\frac{\alpha \in action\ s \qquad s' \in val\ \beta\ s}{(Request\ action\ val,\ s) \rightarrow_{\ll\alpha,\ \beta\gg} (SKIP,\ s')}$$

Response: $$\frac{(s',\ \beta) \in action\ \alpha\ s}{(Response\ action,\ s) \rightarrow_{\gg\alpha,\ \beta\ll} (SKIP,\ s')}$$

LocalOp: $$\frac{s' \in R\ s}{(LocalOp\ R,\ s) \rightarrow_{\tau} (SKIP,\ s')}$$

Seq1: $$\frac{}{(SKIP\ ;;\ c_2,\ s) \rightarrow_{\tau} (c_2,\ s)}$$

Seq2: $$\frac{(c_1,\ s) \rightarrow_{\alpha} (c_1{}',\ s')}{(c_1\ ;;\ c_2,\ s) \rightarrow_{\alpha} (c_1{}'\ ;;\ c_2,\ s')}$$

IfTrue: $$\frac{b\ s \qquad (c_1,\ s) \rightarrow_{\alpha} (c_1{}',\ s')}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \rightarrow_{\alpha} (c_1{}',\ s')}$$

IfFalse: $$\frac{\neg\ b\ s \qquad (c_2,\ s) \rightarrow_{\alpha} (c_2{}',\ s')}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \rightarrow_{\alpha} (c_2{}',\ s')}$$

WhileTrue: $$\frac{b\ s \qquad (c,\ s) \rightarrow_{\alpha} (c',\ s')}{(WHILE\ b\ DO\ c,\ s) \rightarrow_{\alpha} (c'\ ;;\ WHILE\ b\ DO\ c,\ s')}$$

WhileFalse: $$\frac{\neg\ b\ s}{(WHILE\ b\ DO\ c,\ s) \rightarrow_{\tau} (SKIP,\ s)}$$

Choose1: $$\frac{(c_1,\ s) \rightarrow_{\alpha} (c_1{}',\ s')}{(c_1 \sqcup c_2,\ s) \rightarrow_{\alpha} (c_1{}',\ s')}$$

Choose2: $$\frac{(c_2,\ s) \rightarrow_{\alpha} (c_2{}',\ s')}{(c_1 \sqcup c_2,\ s) \rightarrow_{\alpha} (c_2{}',\ s')}$$

Note that the generic nature of the `LocalOp` command lets us choose the atomicity of local actions as appropriate for the language. Since we are in a message passing setting, the atomicity of the internal τ actions is not important for the generation of verification conditions.

With the semantics for the sequential part, we can now define composition of sequential processes into systems. For this purpose, we define the global state of a component system as a function from process names, `'proc`, to configurations. The type `'proc` will later be instantiated with a data type that enumerates precisely all process names in the system.

```
type_synonym ('a, 'proc, 'q, 's) global_state =
  "'proc ⇒ (('a, 'q, 's) com × 's)"
```

With this, we can now define an execution step of the overall system, $\rightarrow$, as either any enabled internal τ step of any process, or as a communication step between two processes. We refer to these alternatives as `LocalStep` and `CommunicationStep`, respectively. For a communication step to occur, two different processes p_1 and p_2 must be ready to execute a request/response pair with matching labels α and β. Again, we present the Isabelle/HOL definition's type signature followed the rule alternatives as proof trees for clarity.

```
inductive system_step ::
  "('a, 'proc, 'q, 's) global_state ⇒ ('a, 'proc, 'q, 's) global_state ⇒ bool"
```

$$\frac{gs\ p \rightarrow_{\tau} c' \qquad gs' = gs(p := c')}{gs \rightarrow gs'}$$

$$\frac{gs\ p_1 \rightarrow_{\ll\alpha,\ \beta\gg} c_1' \qquad gs\ p_2 \rightarrow_{\gg\alpha,\ \beta\ll} c_2' \qquad p_1 \neq p_2 \qquad gs' = gs(p_1 := c_1',\ p_2 := c_2')}{gs \rightarrow gs'}$$

To make this execution model more clear, we consider a simple example of two cooperating processes.

```
datatype proc
  = P1
  | P2
```

We provide a single question and answer and give the processes a state consisting of a single boolean value.

```
datatype question = Ping

datatype answer = Pong

type_synonym state = bool
```

To the first process, we give a program text that sets an initial state then loops sending the question `Ping`.

```
definition
  p1 :: "(answer, question, state) com"
where
  "p1 ≡ LocalOp (λ_. {False}) ;;
        WHILE (λs. ¬ s)
          DO Request (λ_. {Ping}) (λ_ _. {True})"
```

The second process' program text loops responding to messages with `Pong`.

```
definition
  p2 :: "(answer, question, state) com"
where
  "p2 ≡ LOOP Response (λ_ s. {(s, Pong)})"
```

We summarise the execution of these two processes in Figure 5.2. Observe that the system as a whole makes either local steps, in which one process performs an action, or communication steps, in which both processes perform an action. Recall that this model is non-deterministic, so this depiction is a *possible* execution of the system.

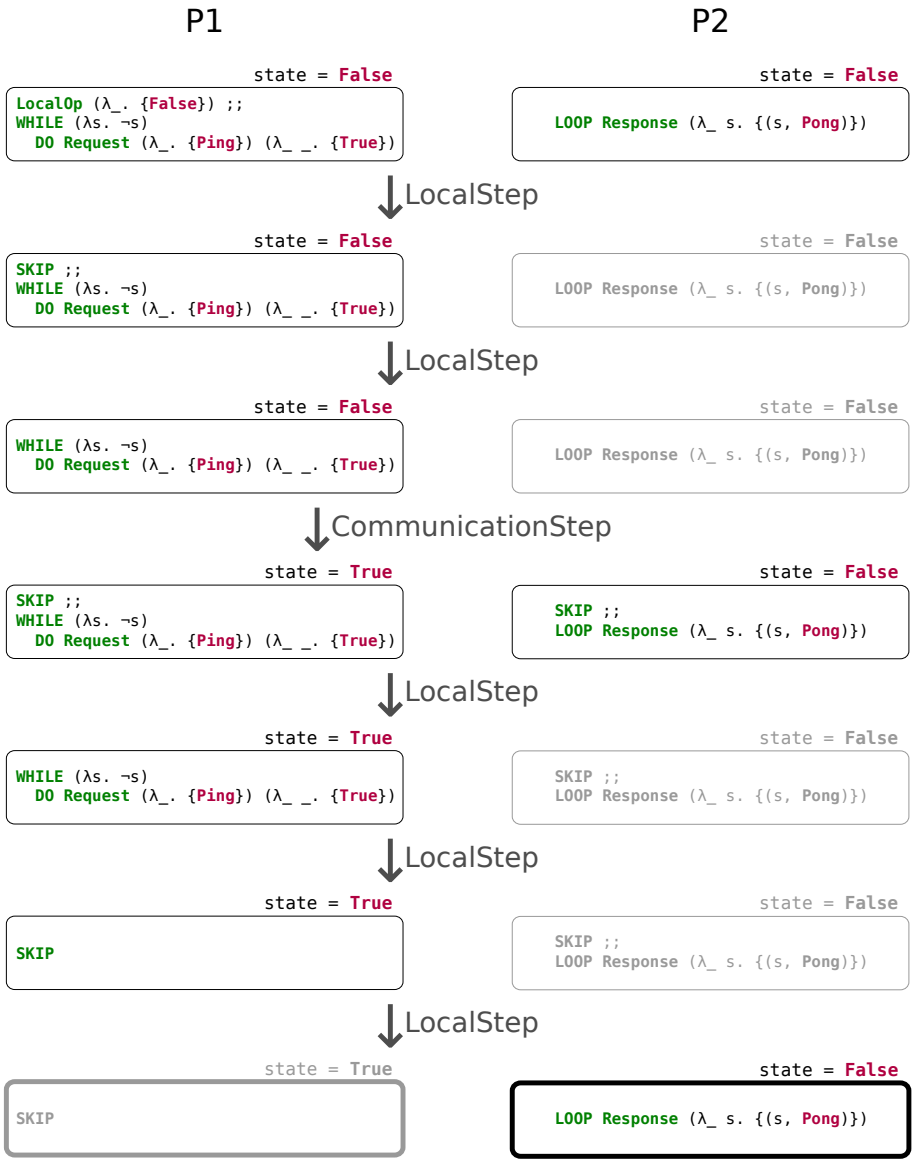

Figure 5.2: Example execution of two CIMP processes

From this point, we could go on to provide standard definitions of finite and infinite execution traces and properties on these, depending on which flavour of properties are desired for a specification verification (for example, invariants, safety, liveness). For the purposes of defining the semantics of CAmkES' glue code we only need the one-step execution, and can therefore leave open which expressive power is desired on top of this semantics structure.

5.3 Type System

This section builds up the basic data types that are necessary to model CAmkES glue code in terms of the concurrent imperative language. In particular, we define data types for the types of variables that glue code interacts with, the type of messages that CAmkES components send and receive, the local state of components, the resulting type of components and finally the partially instantiated, but still generic, global state of a component system.

5.3.1 Messages

Processes communicate via messages which represent IPC payloads in seL4. The only message operations performed in a CAmkES system are initiated by the glue code. Variable data contained in messages are represented using the following data type. This is conceptually equivalent to `param_type` from Chapter 4, with a value attached. Note that there is no need to distinguish the C-specific types in this model, as we are not concerned with the implementation details of the glue code.

```
datatype variable
  = Boolean bool
  | Char char
  | Integer int
  | Number nat
  | String string
  | Array "variable list"
```

Messages are sent from one process to another as questions and acknowledged with answers. Communication with function call semantics – "procedures" in CAmkES terminology – is represented by a sequence of two transmissions; a call and the return. The call message takes a `nat` parameter that is an index indicating which method of the relevant procedure is being invoked. The variable list of a call message contains all the input parameters, while the variable list of a return message contains the return value, if there is one, and the output parameters. To emphasise this, both the call and return of an RPC connection are *questions*, neither are answers, contrary to what may have been anticipated.

Event and shared memory connections are modelled using an intermediate component. This is expanded upon in Section 5.5.

```
datatype question_data
    -- "Inter-component questions"
  = Call nat "variable list"
  | Return "variable list"
    -- "Questions from components to events"
  | Set
  | Poll
    -- "Questions from components to shared memory"
  | Read nat
  | Write nat variable
```

```
datatype answer_data
    -- "Answers from events to components"
  = Pending bool
    -- "Answers from shared memory to components"
  | Value variable
    -- "An answer that conveys no information"
  | Void
```

```
record ('channel) question =
  q_channel :: 'channel
  q_data :: question_data
```

```
record ('channel) answer =
  a_channel :: 'channel
  a_data :: answer_data
```

Message transmission is accomplished using a matching pair of `Request` and `Response` actions. This correspondence arises from using the same channel in a `question` and `answer`. A channel in this representation corresponds to a connection in the CAmkES implementation.

5.3.2 Local State

In this section, we define the local state of components. There are three kinds of components: user-defined, event buffers and shared memory.

We keep the local state of a component parameterised to allow the user to represent as much (or as little) of the concrete state of a component as appropriate for a specific verification. If the local state of a component is not relevant to our current aim, it can be instantiated with an undefined type declaration, as we will demonstrate later.

As mentioned, communication channels involving events and shared memory are modelled using an intermediate component with its own local state. For events, the intermediate component has a single bit of state indicating whether there is a pending signal or not. This is consistent with the desired semantics of the implementation, that emitting an event that is already pending has no effect.

The local state of a shared memory component is a mapping from address offsets (or indices) to variable values. At this level of abstraction, every shared memory region is considered infinite and all operations on the region are represented as manipulations of well-defined types. There is no loss of expressiveness here as raw byte accesses can be represented by mapping each offset to a `variable` of subtype `Number`.

```
datatype 'component_state local_state
  = Component 'component_state
  | Event bool
  | Memory "(nat, variable) map"
```

5.3.3 Components

We model each component in the system as a process. The type itself is only partially instantiated to let the type representing the local state of a component be stated more precisely later as described above.

```
type_synonym ('channel, 'component_state) comp =
  "('channel answer, 'channel question, 'component_state local_state) com"
```

5.3.4 Global State

The global state of a system is a mapping from component instance identifiers to a pair of component (that is, program text) and local state. The global state and local state types are parameterised with general types so they can be instantiated to specifically apply to a given system. During generation, a global state is derived that covers all component instances; that is, the generated global state is total.

```
type_synonym ('inst, 'channel, 'component_state) global_state =
  "('inst, ('channel, 'component_state) comp ×
    'component_state local_state) map"
```

5.4 Convenience Definitions

This section defines static functionality that the generated glue code semantics relies on. It provides the basic building blocks for the CAmkES communication mechanisms. They can also be used as building blocks for users describing the behaviour of trusted components.

5.4.1 Local Component Operations

The set of all possible states a component can be in is referred to as $\texttt{UNIV}_c$. This is essentially any local state with the exception of the states representing events and shared memory. $\texttt{UNIV}_c$ can be thought of as a superset of user component states, or a safe upper bound on the state of a component. That is, a component can be underspecified to have a state that lies in this set.

```
definition
  "UNIV_c" :: "'component_state local_state set"
where
  "UNIV_c ≡ {x. case x of Component _ ⇒ True | _ ⇒ False}"
```

An internal step is one in which a component arbitrarily modifies its own local state. Note that it is not possible for an event or shared memory component (see Section 5.5) to take this step. Analogously to $\texttt{UNIV}_c$, a component's actions that are not observable to other components can be over-approximated by `internal`.

```
definition
  internal :: "'component_state local_state ⇒ 'component_state local_state set"
where
  "internal s ≡ case s of Component _ ⇒ UNIV_c | _ ⇒ {}"
```

We need a wrapper around `internal` that lifts it into the concurrent imperative language. This represents an arbitrary local step.

```
definition
  UserStep :: "('channel, 'component_state) comp"
where
  "UserStep ≡ LocalOp internal"
```

5.4.2 Communication Component Operations

The following operation receives a `Void` message and does nothing in reaction to it. This construct is essentially an artefact of the synchronous message passing basis of our formalism. There are natural actions for a CAmkES component instance to take that represent unidirectional communication, but appear in this formalism as bidirectional. To alleviate this discrepancy, we represent such scenarios using a `discard` action to describe how the sender deals with the (empty) response to their communication. Usage of this can be seen in `EventEmit` below.

```
definition
  discard :: "'channel answer ⇒ 'component_state local_state ⇒
    'component_state local_state set"
where
  "discard a s ≡ if a_data a = Void then {s} else {}"
```

The following expresses non-deterministically sending any message on a given channel. This provides a way of specifying unconstrained behaviour of a component given access to a particular outgoing channel. The command produces the set of all messages on a given channel as possible questions and receives any response with a fully non-deterministic local state update.

```
definition
  ArbitraryRequest :: "'channel ⇒ ('channel, 'component_state) comp"
where
  "ArbitraryRequest c ≡ Request (λ_. {x. q_channel x = c}) (λ_ _. UNIV_c)"
```

The following expresses non-deterministically receiving any message on a given channel. The command receives any message, makes a non-deterministic local state update, and returns the set of all possible responses, β, on the given channel. It is the counterpart of `ArbitraryRequest`, expressing unconstrained behaviour given access to an incoming channel.

```
definition
  ArbitraryResponse :: "'channel ⇒ ('channel, 'component_state) comp"
where
  "ArbitraryResponse c ≡
    Response (λ_ _. {(s',β). s' ∈ UNIV_c ∧ a_channel β = c})"
```

The following command sends the message `Set` on the given channel and discards any response to model asynchronous behaviour with respect to the event buffer components. Note that the message is independent of the local state, `s`. This action is only expected to be used in communication with event components, described in Section 5.5.1.

```
definition
  EventEmit :: "'channel ⇒ ('channel, 'component_state) comp"
where
  "EventEmit c ≡ Request (λs. {(|q_channel = c, q_data = Set|)}) discard"
```

On the receiving side of an event connection, the user has the following primitive for polling for a pending event from an event component. The command sends a `Poll` message to the buffer component, and expects a message, `a`, back that has the form `Pending b` with a boolean payload, `b`. The payload is embedded in the local state of the component using the user-provided function, `embed_data`.

```
definition
  EventPoll :: "'channel ⇒
    ('component_state local_state ⇒ bool ⇒ 'component_state local_state) ⇒
    ('channel, 'component_state) comp"
where
  "EventPoll c embed_data ≡
    Request (λ_. {(|q_channel = c, q_data = Poll|)})
      (λa s. case a_data a of Pending b ⇒ {embed_data s b} | _ ⇒ {})"
```

Similar to `EventPoll` but with blocking semantics, `EventWait` waits for a pending event. The command takes a channel, `c`, and embedding function, `embed_data`. Because the set of target states is only non-empty when the pending bit of the polled event is set, this has the effect of blocking the component's execution until the event is available.

```
definition
  EventWait :: "'channel ⇒
    ('component_state local_state ⇒ bool ⇒ 'component_state local_state) ⇒
    ('channel, 'component_state) comp"
where
  "EventWait c embed_data ≡
    Request (λ_. {|q_channel = c, q_data = Poll|})
      (λa s. case a_data a of Pending b ⇒ if b then {embed_data s b} else {}
                            | _ ⇒ {})"
```

To express reading from a shared memory location we have `MemoryRead`. As mentioned above, shared memory is modelled as a separate process in our glue code semantics. The command below issues a `Read` request message to this process with a specified address, and expects an immediate response in the form `Value v` back, which is embedded into the local state with the same mechanism as above.

```
definition
  MemoryRead :: "'channel ⇒
    ('component_state local_state ⇒ nat) ⇒
    ('component_state local_state ⇒ variable ⇒ 'component_state local_state) ⇒
    ('channel, 'component_state) comp"
where
  "MemoryRead c addr embed_data ≡
    Request (λs. {|q_channel = c, q_data = Read (addr s)|})
      (λa s. case a_data a of Value v ⇒ {embed_data s v} | _ ⇒ {})"
```

The pair to `MemoryRead` is `MemoryWrite` for writing to a shared memory location. The command sends a `Write` message to the memory component with specified address and value (which are extracted from the local state) and does not expect a response.

```
definition
  MemoryWrite :: "'channel ⇒ ('component_state local_state ⇒ nat) ⇒
    ('component_state local_state ⇒ variable) ⇒
    ('channel, 'component_state) comp"
where
  "MemoryWrite c addr val ≡
    Request (λs. {|q_channel = c, q_data = Write (addr s) (val s)|}) discard"
```

5.5 Connector Components

As mentioned in previous sections, we represent events and shared memory as components. These connector components, unlike the component instances in the system, *always* have a well-defined, constrained execution because they are effectively implemented by the kernel. This section outlines the definition of the program text and local state of these components.

The semantics of small steps in the concurrent imperative language are such that a request and a response can only correspond and allow a global state transition when the question and answer match. Additionally, all communication between component instances and connector components is atomic, in the sense that they

involve a single global transition consisting of a single request-response pair. The implication of this is that an untrusted component cannot disrupt the execution of an event or shared memory component causing it to stop responding to other components. Untrusted component definitions may contain unsafe transitions involving malformed messages, but these transitions can never be taken in a global step because there is no corresponding unsafe step in the connector component definition. This faithfully represents the implementation constraints that prevent untrusted components from interfering with the operation of the kernel's mechanisms.

5.5.1 Event Components

We represent a CAmkES event connector as a component always listening for `Set` or `Poll` questions that then immediately responds with the relevant answer. In particular, the local state is expected to be of the form `Event s`, and the component listens to messages of the form `Set` or `Poll`. No other messages are enabled. If a `Set` is received, the local state becomes `Event True`, and the response back is `Void`. If the message is `Poll`, the local event state is cleared, and the response message contains the previous event state, `s`. Recall that waiting on an event is mapped onto a `Poll` message, so there is no need for an extra case to represent this.

```
definition
  event :: "'channel ⇒ ('channel, 'component_state) comp"
where
  "event c ≡ LOOP
    Response (λq s'. case s' of Event s ⇒
      (case q_data q of
         Set ⇒ {(Event True, (|a_channel = q_channel q, a_data = Void|))}
       | Poll ⇒ {(Event False, (|a_channel = q_channel q, a_data = Pending s|))}
       | _ ⇒ {}))"
```

An event component always starts without a pending event. This accurately models the initial state of the underlying asynchronous endpoint.

```
definition
  init_event_state :: "'component_state local_state"
where
  "init_event_state ≡ Event False"
```

5.5.2 Shared Memory Components

We represent shared memory as an always listening component that reads or writes information into its local state. Executing these reads and writes unconditionally accurately represents the behaviour of a read/write region of memory. The implementation is similar to `event`, it merely replaces a one-bit buffer with a map.

```
definition
  memory :: "'channel ⇒ ('channel, 'component_state) comp"
where
  "memory c ≡ LOOP
   Response (λq s'. case s' of Memory s ⇒
    (case q_data q of
      Read addr ⇒ {(Memory s,
                    (|a_channel = q_channel q, a_data = Value (case s addr of
                         Some var ⇒ var
```

```
                        | _ ⇒ Number 0)⦈)}
    | Write addr val ⇒ {(Memory (s(addr ↦ val)),
                         ⦇a_channel = q_channel q, a_data = Void⦈)})
    | _ ⇒ {})"
```

The initial state of a shared memory component is an empty map. A shared memory region is assumed to be zeroed on startup. Note that the preceding definition guarantees these semantics.

```
definition
  init_memory_state :: "'component_state local_state"
where
  "init_memory_state ≡ Memory empty"
```

In CAmkES descriptions, shared memory regions can have a type, typically defined as a C struct. For now, only the default type `Buf` is represented in this model. The model can be trivially extended to represent user types as components with a memory local state that have additional constraints on what can be read from or written to the state.

```
type_synonym "Buf_d_channel" = unit

definition
  "Buf_d" :: "(Buf_d_channel ⇒ 'channel) ⇒ ('channel, 'component_state) comp"
where
  "Buf_d ch ≡ memory (ch ())"
```

5.6 Component Behaviour

A combination of generated and user-provided theories provide the definitions of a full system. CAmkES creates a base theory using the types and definitions previously discussed that defines primitive operations of a specific system. The system designer is then expected to provide a theory that constrains the system behaviour by giving more precise definitions for the trusted component instances. We then automatically extract a generated state that contains definitions for each component instance, falling back on the generated, generalised definitions for any instance for which a trusted definition has not been provided. The designer can then perform correctness proofs on this generated state.

This interleaving of generated and user-provided theories is achieved using a *locale*. A locale is an Isabelle/HOL feature designed to capture a set of fixed parameters and assumptions that apply to all enclosed proofs. The locale we use in this formalism has the effect of leaving "holes" in the generated base definitions of a system for the type of component instances' state and the definitions of trusted component instances, that will be filled in by the system designer post hoc. Appendix B provides a more extended introduction to the concept of locales for the interested reader.

The theory interleaving we have described may appear counterintuitive but leads to an environment that balances expressivity with automation. The designer is free to specify as much or as little of their components' behaviour as they wish, leaving unspecified behaviour to fall back on a generic upper bound provided by the platform.

If the system designer's desired correctness property is provable using the generalised component instance definitions, they are free to provide no trusted component instance definitions at all. However, any such property will only be provable by way of architectural constraints of the system. A simpler approach for proving such properties is to use the formalism from Chapter 4, wherein such proofs would be automatic.

When using generated theories from the current formalism, the designer is expected to provide the following type instantiations and definitions:

- A type for `component_state` indicating the local state that should be represented for each component;
- An initial `component_state` for untrusted components to be given on startup; and
- A (possibly empty) mapping from component instance identifiers to trusted component definitions.

This relationship between generated and user-provided artefacts is depicted in Figure 5.3.

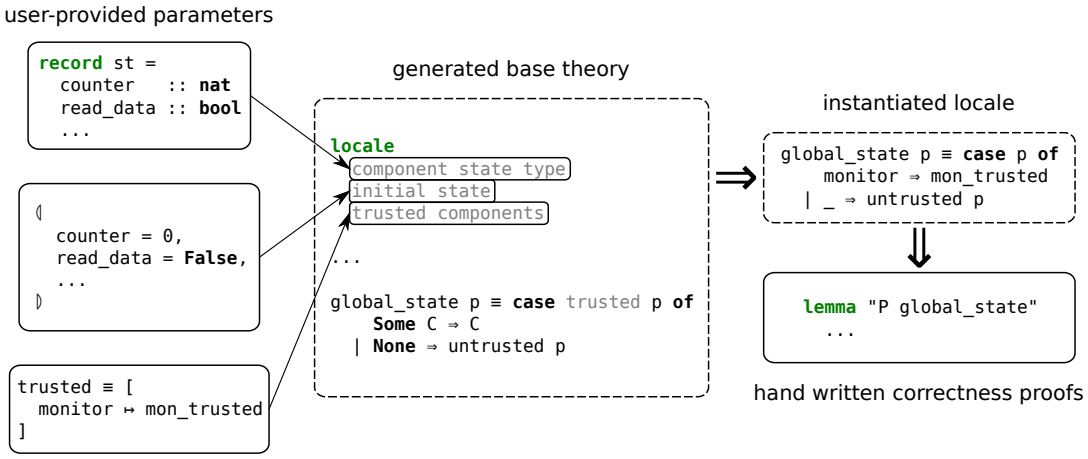

Figure 5.3: Instantiating a generated locale with user-provided inputs

The remainder of this section describes the default component definitions that are fallen back on when trusted definitions are not provided.

5.6.1 Local Component State

The user should specify a type for `component_state` if they want to reason about the behaviour of user-provided code. If not, then they can use a type declaration to create a new, undefined type.

```
1 typedecl component_state
```

The generated theories need to be provided with a default value for the local state type. This is used as the initial local state for untrusted components. This definition must be visible even if there are no untrusted components in the system, although in this case it will not be used. If the components' state is irrelevant to the current aim, the user can provide a constant of the type declared previously. Note that nothing is known about this term.

```
1 consts init_component_state :: component_state
```

5.6.2 Untrusted Component Instances

Any component which does not have its definition supplied will be given a generated definition that allows it to non-deterministically perform any local operations or send or receive anything on any channel available to it. Without providing definitions of any trusted components it will generally be impossible to reason about anything beyond architectural properties of the system.

5.6.3 Trusted Component Instances

Trusted components should be given a defined program text (type `component`) and an initial local state. The user should provide a definition of `trusted`, a mapping from component instances to a pair of component and initial local state. Any instance not present in the mapping will be assigned the broad definition described in the previous paragraph.

The default mapping is as defined below, empty, causing all instances to fall back on their untrusted definitions.

```
type_synonym ('channel) component = "('channel, component_state) comp"
```

```
type_synonym lstate = "component_state local_state"
```

```
definition
  trusted :: "('inst, ('channel component × lstate)) map"
where
  "trusted ≡ empty"
```

5.7 Example – Procedures

In this section, we provide an example of the generated types and definitions that are derived from a CAmkES procedure interface. Throughout, this section uses an example system involving two component instances defined by the following CAmkES specification.

```
procedure Simple {
  int echo_int(in int i);
  int echo_parameter(in int pin, out int pout);
}

component Client {
  control;
  uses Simple s;
}

component Echo {
  provides Simple s;
}

assembly {
  composition {
    component Echo echo;
    component Client client;

    connection seL4RPCSimple simple(from client.s, to echo.s);
  }
}
```

The system can be depicted as two component instances connected with a single interface in Figure 5.4.

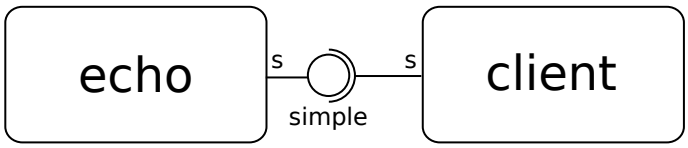

Figure 5.4: Simple component system

5.7.1 Types

Data types are generated to enumerate the connections in the system, `channel`, and the component instances in the system, `inst`. As this system only has a single connection, the `channel` data type is trivial. Note that the `inst` type enumerates component *instances*, not component *types*.

```
datatype channel
  = simple
```

```
datatype inst
  = client
  | echo
```

For each component type, a data type is generated that enumerates the interfaces of that component.

```
datatype Client_channel
  = Client_s
```

```
datatype Echo_channel
  = Echo_s
```

This type does not indicate whether the interfaces are outgoing or incoming, or what type of interface they represent. All component type definitions are parameterised with a mapping from this type to `channel`. When a component type is instantiated, this mapping must be specified to describe the architecture of the system. In this example system, both component instances each have their single interface mapped to the only connection, `simple`.

5.7.2 Instantiations of Primitives

The component instance primitives are relatively straightforward, as they are formed by partially applying component interface definitions, that we will introduce in the next section. We show the definition of the instance primitives first to aid understanding of the motivation for the interface primitives that are quite dense.

A call definition is generated for each method in each outgoing interface in each component instance that partially applies the call definitions from the next section, with a mapping from the component's interface to the system connection.

The parameter used to specialise the component interface primitives, a function from that component's channel type to the system channel type, is derived from the architecture of the system. In this example the instance

`client` has its interface `s` connected to the connection `simple`. Thus its primitives are expressed using a function that maps its channel type `Client_s` to the corresponding system channel, `simple`. In the case of this example where the `client` instance has a single interface, the function could be given as $\lambda\_.$ `simple`, but for simplicity the generator does not make this optimisation.

```
definition
  Call_client_s_echo_int :: "('cs local_state ⇒ int) ⇒
    ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Call_client_s_echo_int ≡
    Call_Client_s_echo_int (λc. case c of Client_s ⇒ simple)"

definition
  Call_client_s_echo_parameter :: "('cs local_state ⇒ int) ⇒
    ('cs local_state ⇒ int ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Call_client_s_echo_parameter ≡
    Call_Client_s_echo_parameter (λc. case c of Client_s ⇒ simple)"
```

Similarly, a receive definition is generated for each incoming interface in each component instance.

```
definition
  Recv_echo_s :: "('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp ⇒ ('cs local_state ⇒ int) ⇒
    ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp ⇒ ('cs local_state ⇒ int) ⇒
    ('cs local_state ⇒ int) ⇒ (channel, 'cs) comp"
where
  "Recv_echo_s ≡ Recv_Echo_s (λc. case c of Echo_s ⇒ simple)"
```

Recall that we have a receive definition per-*interface*, not per-method, as it performs method dispatch.

5.7.3 Interface Primitives

We now present the definitions of the interface primitives, the functions used in the previous section. We describe the glue code specifications generated for each interface of each component type. For an outgoing procedure interface, a definition is generated for each method in that interface, prefixed by "Call," the component name and the interface name. These can be composed with each other and user-provided steps to form a process that describes the execution of the component.

The interface in this example has two methods, `echo_int` and `echo_parameter`, hence two separate call definitions are generated. These functions take a sequence of embedding and projection functions into and out of the component's local state. The types of these functions are derived from the parameter types of the method and are used for marshalling arguments. For example, `echo_int` takes an `int` input parameter, `i`, which necessitates a projection function, $\texttt{i}_\texttt{P}$, as a parameter to `Call_Client_s_echo_int`. Conversely, the method returns an `int` parameter, necessitating an embedding function, `embed_data` as a parameter. In general, an input parameter requires a projection, an output parameter or return value requires an embedding and an input/output parameter requires both.

```
definition
  Call_Client_s_echo_int :: "(Client_channel ⇒ channel) ⇒
    ('cs local_state ⇒ int) ⇒
    ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Call_Client_s_echo_int ch iₚ embed_data ≡
    Request (λs. {(|q_channel = ch Client_s,
      q_data = Call 0 [Integer (iₚ s)]|)}) discard ;;
    Response (λq s. case q_data q of Return xs ⇒
      {(embed_data s (case hd xs of Integer v ⇒ v),
      (|a_channel = ch Client_s, a_data = Void|))} | _ ⇒ {})"

definition
  Call_Client_s_echo_parameter :: "(Client_channel ⇒ channel) ⇒
    ('cs local_state ⇒ int) ⇒
    ('cs local_state ⇒ int ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Call_Client_s_echo_parameter ch pinₚ embed_data ≡
    Request (λs. {(|q_channel = ch Client_s,
      q_data = Call 1 [Integer (pinₚ s)]|)}) discard ;;
    Response (λq s. case q_data q of Return xs ⇒
      {(embed_data s (case hd xs of Integer v ⇒ v)
         (case xs ! 1 of Integer v ⇒ v),
      (|a_channel = ch Client_s, a_data = Void|))} | _ ⇒ {})"
```

For an incoming procedure interface, a single definition is generated with the prefix "Recv," the component's name and the interface name. There is a single definition on the incoming side, rather than one per interface, to match the semantics of the implementation. That is, the blocking receive followed by method dispatch are captured in this definition. Projection and embedding functions are again necessitated, but their roles are reversed.

Each receive definition is also parameterised with a process for each method representing the user-provided implementation of this method. For example, in the definition below, the `Echo_s_echo_int` parameter is expected to be the user's implementation of the `echo_int` method.

```
definition
  Recv_Echo_s :: "(Echo_channel ⇒ channel) ⇒
    ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp ⇒ ('cs local_state ⇒ int) ⇒
    ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp ⇒ ('cs local_state ⇒ int) ⇒
    ('cs local_state ⇒ int) ⇒ (channel, 'cs) comp"
where
  "Recv_Echo_s ch echo_intₑ Echo_s_echo_int echo_int_returnₚ
     echo_parameterₑ Echo_s_echo_parameter echo_parameter_returnₚ
     echo_parameter_poutₚ ≡
    (Response (λq s. case q_data q of Call n xs ⇒
      (if n = 0 then {(echo_intₑ s (case xs ! 0 of Integer v ⇒ v),
        (|a_channel = ch Echo_s, a_data = Void|))} else {}) | _ ⇒ {}) ;;
     Echo_s_echo_int ;;
     Request (λs. {(|q_channel = ch Echo_s,
       q_data = Return [Integer (echo_int_returnₚ s) ]|)}) discard)
    ⊔
    (Response (λq s. case q_data q of Call n xs ⇒
      (if n = 1 then {(echo_parameterₑ s (case xs ! 0 of Integer v ⇒ v),
```

```
21          (|a_channel = ch Echo_s, a_data = Void|)} else {}) | _ ⇒ {}) ;;
22       Echo_s_echo_parameter ;;
23       Request (λs. {(|q_channel = ch Echo_s,
24         q_data = Return [Integer (echo_parameter_return_P s),
25                          Integer (echo_parameter_pout_P s)]|)}) discard)"
```

Though the type signature and definition of this term are extensive, observe that they are not complex. The large number of parameters is motivated by the manner in which this generated definition must interact with user-provided definitions.

5.7.4 Untrusted Components

For each component type, a definition is generated that describes its execution without specifying the behaviour of any user-provided code. These definitions allow the component to perform any manipulation of its local state or to send or receive any message on the interfaces available to it. These definitions are intended for use in a system composition when the behaviour of a specific component is not relevant to the desirable property of the whole system. These definitions are more general than the implementation allows. For example, they permit an untrusted component to perform actions such as sending on an incoming interface which may not be possible in the implementation. This over-generalisation is deliberate, as we want to capture an upper bound on component behaviour.

Recall from Section 5.6 that the user is expected to provide a mapping describing trusted component instances. A definition of untrusted execution for each component instance is generated unconditionally as the platform does not know in advance what property the user ultimately wishes to prove and hence does not know which instances will need trusted definitions.

```
1  definition
2    Client_untrusted :: "(Client_channel ⇒ channel) ⇒ (channel, 'cs) comp"
3  where
4    "Client_untrusted ch ≡
5      LOOP (
6        UserStep
7      ⌴ ArbitraryRequest (ch Client_s)
8      ⌴ ArbitraryResponse (ch Client_s))"
9
10 definition
11   Echo_untrusted :: "(Echo_channel ⇒ channel) ⇒ (channel, 'cs) comp"
12 where
13   "Echo_untrusted ch ≡
14     LOOP (
15       UserStep
16     ⌴ ArbitraryRequest (ch Echo_s)
17     ⌴ ArbitraryResponse (ch Echo_s))"
```

5.7.5 Component Instances

As was the case for the instantiation of primitives in Section 5.7.2, with the definition of an untrusted component's execution generated previously, a definition of the execution of an untrusted instance can be formed by partially applying the component definition. A definition of untrusted execution is generated for each component instance, whether it is required or not.

```
definition
  client_untrusted :: "(channel, 'cs) comp"
where
  "client_untrusted ≡ Client_untrusted (λc. case c of Client_s ⇒ simple)"

definition
  echo_untrusted :: "(channel, 'cs) comp"
where
  "echo_untrusted ≡ Echo_untrusted (λc. case c of Echo_s ⇒ simple)"
```

5.7.6 Initial State

The final generated definition is the initial state of the system. Following the type instantiations in Section 5.7.1, the initial global state is a mapping from component instance names to a pair of their program text and local state. The generated definition looks for the instance's definition in the mapping of trusted instances and, if it does not find this, falls back on the generated untrusted definitions. Recall that the mapping of trusted instances is a locale parameter to be provided by the designer later.

```
definition
  "gs₀" :: "(inst, channel, 'cs) global_state"
where
  "gs₀ p ≡ case trusted p of Some s ⇒ Some s | _ ⇒
  (case p of client ⇒ Some (client_untrusted, Component init_component_state)
           | echo ⇒ Some (echo_untrusted, Component init_component_state))"
```

5.8 Example – Events

Following on from Section 5.7, this section gives a corresponding example of the definitions that are generated for a system involving CAmkES events. A system defined by the following specification will be used throughout.

```
component Emitter {
  control;
  emits SomethingHappenedEvent ev;
}

component Collector {
  control;
  consumes SomethingHappenedEvent ev;
}

assembly {
  composition {
    component Emitter source;
    component Collector sink;

    connection seL4AsynchNative simpleEvent1(from source.ev, to sink.ev);
  }
}
```

5.8.1 Types

The data types generated for a system involving events are similar to those for a system involving procedures, however an additional instance is derived for every connection in the system that carries event messages. This generated instance models the state of the event; that is, whether it is pending or not.

```
datatype channel
  = simpleEvent1

datatype inst
  = sink
  | source
  | "simpleEvent1ₑ" (* artificial component to represent event *)

datatype Collector_channel
  = Collector_ev

datatype Emitter_channel
  = Emitter_ev
```

5.8.2 Interface Primitives

For each component type with a `consumes` interface, two primitives are generated for each interface. These correspond to the wait and poll functions in generated glue code. As for procedures, `embed_data` functions must be supplied by the designer to save the result of the operation into the component's local state.

```
definition
  Poll_Collector_ev :: "(Collector_channel ⇒ channel) ⇒
    ('cs local_state ⇒ bool ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Poll_Collector_ev ch embed_data ≡ EventPoll (ch Collector_ev) embed_data"

definition
  Wait_Collector_ev :: "(Collector_channel ⇒ channel) ⇒
    ('cs local_state ⇒ bool ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Wait_Collector_ev ch embed_data ≡ EventWait (ch Collector_ev) embed_data"
```

For each component type with an `emits` interface, a single primitive is generated to correspond to the emit function in the glue code. The emit definition needs no embedding or projection functions because it is state-independent.

```
definition
  Emit_Emitter_ev :: "(Emitter_channel ⇒ channel) ⇒ (channel, 'cs) comp"
where
  "Emit_Emitter_ev ch ≡ EventEmit (ch Emitter_ev)"
```

5.8.3 Instantiations of Primitives

As for procedure interfaces, the event primitives are specialised for each interface in the system by partially applying them with a function mapping the interface to the relevant – in this case the only – system connection.

```
definition
  Poll_sink_ev :: "('cs local_state ⇒ bool ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Poll_sink_ev ≡
    Poll_Collector_ev (λc. case c of Collector_ev ⇒ simpleEvent1)"

definition
  Wait_sink_ev :: "('cs local_state ⇒ bool ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Wait_sink_ev ≡
    Wait_Collector_ev (λc. case c of Collector_ev ⇒ simpleEvent1)"

definition
  Emit_source_ev :: "(channel, 'cs) comp"
where
  "Emit_source_ev ≡
    Emit_Emitter_ev (λc. case c of Emitter_ev ⇒ simpleEvent1)"
```

5.8.4 Untrusted Components

As before, an untrusted definition is generated for each component type that permits any local operation or sending or receiving on any available interface. Once again, we deliberately over-approximate the components' behaviour.

```
definition
  Collector_untrusted :: "(Collector_channel ⇒ channel) ⇒ component"
where
  "Collector_untrusted ch ≡
    LOOP (
      UserStep
    ⊔ ArbitraryRequest (ch Collector_ev)
    ⊔ ArbitraryResponse (ch Collector_ev))"

definition
  Emitter_untrusted :: "(Emitter_channel ⇒ channel) ⇒ component"
where
  "Emitter_untrusted ch ≡
    LOOP (
      UserStep
    ⊔ ArbitraryRequest (ch Emitter_ev)
    ⊔ ArbitraryResponse (ch Emitter_ev))"
```

5.8.5 Event Components

For each connection in the system over which events are transmitted, a definition is generated of a component type that models the state of the event. The type enumerating the interfaces of this component is expressed as `unit` because, naturally, there is only a single interface to this introduced component. The details of the execution of the component can largely be expressed statically, and are captured by the definition, `event`, given in Section 5.5.1.

```
type_synonym SomethingHappenedEvent_channel = unit

definition
  SomethingHappenedEvent :: "(SomethingHappenedEvent_channel ⇒ channel) ⇒
    component"
where
  "SomethingHappenedEvent ch ≡ event (ch ())"
```

5.8.6 Component Instances

The definitions of untrusted component instances are generated as in Section 5.7.5, but a definition is also derived for an instance of the introduced component. There is no opportunity for the user to provide a definition of the trusted execution of this component, because we already know exactly what actions this component takes. Being part of the component platform itself, we can generate a definition that exactly expresses its execution.

```
definition
  sink_untrusted :: component
where
  "sink_untrusted ≡
    Collector_untrusted (λc. case c of Collector_ev ⇒ simpleEvent1)"

definition
  source_untrusted :: component
where
  "source_untrusted ≡
    Emitter_untrusted (λc. case c of Emitter_ev ⇒ simpleEvent1)"

definition
  "simpleEvent1ₑ_instance" :: component
where
  "simpleEvent1ₑ_instance ≡ SomethingHappenedEvent (λ_. simpleEvent1)"
```

5.8.7 Initial State

The generated global state for this system also contains a case for the introduced event component, mapping to the instance definition presented above and the common initial event state. While this definition of the global state makes it possible for the user to override the mapping of `simpleEvent1`$_e$ in `trusted`, there is no practical reason to do this.

```
definition
  "gs₀" :: gstate
where
  "gs₀ p ≡ case trusted p of Some s ⇒ Some s | _ ⇒
  (case p of sink ⇒ Some (sink_untrusted, Component init_component_state)
           | source ⇒ Some (source_untrusted, Component init_component_state)
           | simpleEvent1ₑ ⇒ Some (simpleEvent1ₑ_instance, init_event_state))"
```

5.9 Example – Dataports

The final example we provide is the corresponding set of generated types and definitions derived from a CAmkES dataport interface. The specification we will use in this case is as follows.

```
component DataportTest {
  control;
  dataport Buf d1;
  dataport Buf d2;
}

assembly {
  composition {
    component DataportTest comp1;
    component DataportTest comp2;

    connection seL4SharedData simple1(from comp1.d1, to comp2.d2);
    connection seL4SharedData simple2(from comp2.d1, to comp1.d2);
  }
}
```

Note that this system, unlike the previous two examples, contains a component type that is instantiated twice.

5.9.1 Types

As with the previous examples, a type is generated for the connections in the system and the component instances in the system. The data type, `channel`, is as before, but `inst` also contains a member generated for each connection in the system involving a dataport, $\texttt{simple1}_\texttt{d}$ and $\texttt{simple2}_\texttt{d}$.

```
datatype channel
  = simple1
  | simple2

datatype inst
  = comp1
  | comp2
  | "simple1_d"
  | "simple2_d"
```

The type for the interfaces of the single component in the system is generated as in the previous examples.

```
datatype DataportTest_channel
  = DataportTest_d1
  | DataportTest_d2
```

5.9.2 Interface Primitives

For each dataport interface of each component type, definitions are generated for performing a read or write to the dataport. Like events, the details of these operations can be determined statically and are captured in the definitions, `MemoryRead` and `MemoryWrite`, introduced previously.

Read and write are unconditionally generated for each dataport interface because all dataports are read/write memory regions. A trivial extension to this formalism would be to model dataports with constrained permissions, but we defer this for now. The motivation for dataport-related compromises is discussed in Section 9.3.

```
definition
  Read_DataportTest_d1 :: "(DataportTest_channel ⇒ channel) ⇒
    ('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ variable ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
 "Read_DataportTest_d1 ch addr embed_data ≡
   MemoryRead (ch DataportTest_d1) addr embed_data"

definition
  Write_DataportTest_d1 :: "(DataportTest_channel ⇒ channel) ⇒
    ('cs local_state ⇒ nat) ⇒ ('cs local_state ⇒ variable) ⇒
    (channel, 'cs) comp"
where
  "Write_DataportTest_d1 ch addr proj ≡
    MemoryWrite (ch DataportTest_d1) addr proj"

definition
  Read_DataportTest_d2 :: "(DataportTest_channel ⇒ channel) ⇒
    ('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ variable ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Read_DataportTest_d2 ch addr embed_data ≡
    MemoryRead (ch DataportTest_d2) addr embed_data"

definition
  Write_DataportTest_d2 :: "(DataportTest_channel ⇒ channel) ⇒
    ('cs local_state ⇒ nat) ⇒ ('cs local_state ⇒ variable) ⇒
    (channel, 'cs) comp"
where
  "Write_DataportTest_d2 ch addr proj ≡
    MemoryWrite (ch DataportTest_d2) addr proj"
```

5.9.3 Instantiations of Primitives

The specialisation of the primitives from Section 5.9.2 is similar to the previous examples, except that multiple instantiations for each are generated because the component type `DataportTest` is instantiated twice in this system.

```
definition
  Read_comp1_d1 :: "('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ variable ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Read_comp1_d1 ≡
    Read_DataportTest_d1 (λc. case c of DataportTest_d2 ⇒ simple2
                                      | DataportTest_d1 ⇒ simple1)"

definition
  Write_comp1_d1 :: "('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ variable) ⇒ (channel, 'cs) comp"
where
```

```
13    "Write_comp1_d1 ≡
14      Write_DataportTest_d1 (λc. case c of DataportTest_d2 ⇒ simple2
15                                         | DataportTest_d1 ⇒ simple1)"
16
17  definition
18    Read_comp1_d2 :: "('cs local_state ⇒ nat) ⇒
19      ('cs local_state ⇒ variable ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
20  where
21    "Read_comp1_d2 ≡
22      Read_DataportTest_d2 (λc. case c of DataportTest_d2 ⇒ simple2
23                                       | DataportTest_d1 ⇒ simple1)"
24
25  definition
26    Write_comp1_d2 :: "('cs local_state ⇒ nat) ⇒
27      ('cs local_state ⇒ variable) ⇒ (channel, 'cs) comp"
28  where
29    "Write_comp1_d2 ≡
30      Write_DataportTest_d2 (λc. case c of DataportTest_d2 ⇒ simple2
31                                         | DataportTest_d1 ⇒ simple1)"
32
33  definition
34    Read_comp2_d1 :: "('cs local_state ⇒ nat) ⇒
35      ('cs local_state ⇒ variable ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
36  where
37    "Read_comp2_d1 ≡
38      Read_DataportTest_d1 (λc. case c of DataportTest_d1 ⇒ simple2
39                                       | DataportTest_d2 ⇒ simple1)"
40
41  definition
42    Write_comp2_d1 :: "('cs local_state ⇒ nat) ⇒
43      ('cs local_state ⇒ variable) ⇒ (channel, 'cs) comp"
44  where
45    "Write_comp2_d1 ≡
46      Write_DataportTest_d1 (λc. case c of DataportTest_d1 ⇒ simple2
47                                         | DataportTest_d2 ⇒ simple1)"
48
49  definition
50    Read_comp2_d2 :: "('cs local_state ⇒ nat) ⇒
51      ('cs local_state ⇒ variable ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
52  where
53    "Read_comp2_d2 ≡
54      Read_DataportTest_d2 (λc. case c of DataportTest_d1 ⇒ simple2
55                                       | DataportTest_d2 ⇒ simple1)"
56
57  definition
58    Write_comp2_d2 :: "('cs local_state ⇒ nat) ⇒
59      ('cs local_state ⇒ variable) ⇒ (channel, 'cs) comp"
60  where
61    "Write_comp2_d2 ≡
62      Write_DataportTest_d2 (λc. case c of DataportTest_d1 ⇒ simple2
63                                         | DataportTest_d2 ⇒ simple1)"
```

5.9.4 Untrusted Components

A definition is generated for the untrusted execution of the component, `DataportTest`. In this definition there are two interfaces the component can send and receive on, but the other details of the definition are identical to the previous examples.

```
definition
  DataportTest_untrusted :: "(DataportTest_channel ⇒ channel) ⇒ component"
where
  "DataportTest_untrusted ch ≡
    LOOP (
      UserStep
    ⌊⌋ ArbitraryRequest (ch DataportTest_d2)
    ⌊⌋ ArbitraryResponse (ch DataportTest_d2)
    ⌊⌋ ArbitraryRequest (ch DataportTest_d1)
    ⌊⌋ ArbitraryResponse (ch DataportTest_d1))"
```

5.9.5 Component Instances

The definitions for untrusted execution of the two component instances are generated by partially applying the untrusted definition of `DataportTest` with different functions mapping its interfaces to connections. In this way, two processes are formed that have identical local behaviour, but have different effects when they perform communication actions.

```
definition
  comp2_untrusted :: component
where
  "comp2_untrusted ≡
    DataportTest_untrusted (λc. case c of DataportTest_d1 ⇒ simple2
                                        | DataportTest_d2 ⇒ simple1)"

definition
  comp1_untrusted :: component
where
  "comp1_untrusted ≡
    DataportTest_untrusted (λc. case c of DataportTest_d2 ⇒ simple2
                                        | DataportTest_d1 ⇒ simple1)"
```

5.9.6 Shared Memory Components

A component instance is generated for each connection involving a dataport, as mentioned previously. As for events, the user is given no opportunity to provide trusted definitions for these instances because we can automatically generate their precise behaviour without ambiguity.

```
definition
  "simple2ₐ_instance" :: component
where
  "simple2ₐ_instance ≡ Bufₐ (λ_. simple2)"

definition
  "simple1ₐ_instance" :: component
where
  "simple1ₐ_instance ≡ Bufₐ (λ_. simple1)"
```

5.9.7 Initial State

The initial state for this system includes cases for the introduced shared memory components, using the definitions presented above. Both begin in the common initial memory state containing the empty map. Again,

the designer could override the definitions of $\texttt{simple1}_\texttt{d}$ and $\texttt{simple2}_\texttt{d}$ in `trusted` but there is no practical reason to do this.

```
definition
  "gs₀" :: gstate
where
  "gs₀ p ≡ case trusted p of Some s ⇒ Some s | _ ⇒
  (case p of comp2 ⇒ Some (comp2_untrusted, Component init_component_state)
           | comp1 ⇒ Some (comp1_untrusted, Component init_component_state)
           | simple2_d ⇒ Some (simple2_d_instance, init_memory_state)
           | simple1_d ⇒ Some (simple1_d_instance, init_memory_state))"
```

5.10 Example – System Level Reasoning

Having introduced the foundations of this model, we now return to the example from Section 5.1. We omit the generated types and definitions for this example as they are similar to those from the previous sections, and instead move to properties that can now be proven about this system.

5.10.1 Architectural Properties

Using the most generalised (untrusted) version of the system, we cannot show anything except architectural properties. These are true by construction of the generated system. To demonstrate this, we show a proof that the `client` and `store` instances cannot directly communicate. For this we need to *interpret* the locale, providing default values for the locale parameters. See Appendix B for a primer on locale interpretation in general. The default parameters we provide are `unit`, `()`, to indicate the components have no relevant local state and an empty map of trusted component instances.

```
interpretation system_state "()" empty
  done
```

First we introduce some definitions to aid the statement of the property. A predicate specifying that a component sends on a given channel is defined as `sends_on`.

```
fun
  sends_on :: "channel ⇒ component ⇒ bool"
where
    "sends_on c (Request f _) = (∃s. ∃q ∈ f s. q_channel q = c)"
  | "sends_on c (a ;; b) = (sends_on c a ∨ sends_on c b)"
  | "sends_on c (IF cond THEN a ELSE b) =
      (∀s. cond s ∧ sends_on c a ∨ ¬ cond s ∧ sends_on c b)"
  | "sends_on c (WHILE cond DO a) = (∀s. cond s ∧ sends_on c a ∨ ¬ cond s)"
  | "sends_on c (a ⊔ b) = (sends_on c a ∨ sends_on c b)"
  | "sends_on _ _ = False"
```

The corresponding predicate for receiving on a channel is defined as `receives_on`.

```
fun
  receives_on :: "channel ⇒ component ⇒ bool"
where
    "receives_on c (Response f) = (∃q s. ∃a ∈ f q s. a_channel (snd a) = c)"
  | "receives_on c (a ;; b) = (receives_on c a ∨ receives_on c b)"
  | "receives_on c (IF cond THEN a ELSE b) =
      (∀s. cond s ∧ receives_on c a ∨ ¬ cond s ∧ receives_on c b)"
  | "receives_on c (WHILE cond DO a) =
      (∀s. cond s ∧ receives_on c a ∨ ¬ cond s)"
  | "receives_on c (a ⊔ b) = (receives_on c a ∨ receives_on c b)"
  | "receives_on _ _ = False"
```

Now whether a component communicates on a channel can be defined as the disjunction of these two.

```
definition
  communicates_on :: "channel ⇒ component ⇒ bool"
where
  "communicates_on ch c ≡ sends_on ch c ∨ receives_on ch c"
```

We can now state, and prove, the property that `client` and `store` never directly communicate. Note that the theorem prover needs to be manually directed to a solution by applying the proof tactic `auto` and supplying definitions to unfold. Were one merely interested in this property, a simpler way to achieve such a guarantee would be an automated proof within the formalism of Chapter 4.

```
lemma "∀c. ¬(communicates_on c client_untrusted ∧
             communicates_on c store_untrusted)"
  by (auto simp:communicates_on_def client_untrusted_def Client_untrusted_def
                store_untrusted_def Store_untrusted_def UserStep_def
                ArbitraryRequest_def ArbitraryResponse_def)
```

Were we to try reasoning about a property of the system that depended upon the behaviour of any component in the system, we would not be able to do it using the existing definitions. To show a property of this form we need to provide a more precise definition of the critical components. An example of this is shown in the next section.

5.10.2 Behavioural Properties

To reason about the behaviour of components themselves, we need to provide more information when interpreting the locale. In this section we present an example of this and a proof that `client` never receives the secret, 42. This property is dependent on the behaviour of `filter`, to which `client` is directly connected.

First we specify a more precise set of messages to be sent by `filter`. We define its valid responses as only the values 7 and -1.

```
definition
  filter_responses :: "channel question set"
where
  "filter_responses ≡ {x. ∃v ∈ {7, -1}. q_data x = Return [Integer v]}"
```

Then we give a more constrained definition of `filter` that no longer allows it to send any message on the channel connected to `client`. Note that for the target property we can still leave the remainder of its behaviour arbitrary.

```
definition
  filter_trusted :: component
where
  "filter_trusted ≡
    LOOP (
      UserStep
    ⨆ ArbitraryRequest two
    ⨆ ArbitraryResponse two
    ⨆ ArbitraryResponse one
    ⨆ Request (λ_. filter_responses) discard)"
```

This trusted definition of `filter` is passed to the generated system theory in the definition of `trusted`.

```
definition
  trusted :: "(inst, (component × lstate)) map"
where
  "trusted ≡ [filter ↦ (filter_trusted, Component init_component_state)]"
```

We can use these definitions to interpret the locale with a different set of constraints.

```
interpretation system_state "()" trusted
  done
```

Now it is possible to state and prove the desired property of the system; that `client` never receives the secret 42.

```
lemma "∀p. ∃e s. gs₀ p = Some (e, s) ∧ (e = client_untrusted ∨
  ¬(∃c. sends e {x. q_channel x = c ∧ q_data x = Return [Integer 42]} ∧
  receives_on c client_untrusted))"
```

While the proof of this property (elided) was relatively straightforward, its link to the implementation remains informal. We manually specified the behaviour of `filter` and there is little to guarantee that this specification matches the actual behaviour of `filter`. In the next chapter we introduce a formalism with a stronger link to the implementation, at the expense of automation. A significant advantage of the formalism of the present chapter is the ease with which properties can be formalised and proven.

5.11 Summary

In this chapter, we have described a formalism for representing component system architectures and the behaviour of component instances. We have demonstrated how generated and user-provided theories interact and how this can be applied to prove sample correctness properties. The guarantee that the user-provided description of a component's behaviour matches the behaviour of their provided code for that component is informal, but this trade off enables significant automation in both system specification and reasoning. By streamlining the process of system specification, we enable an accessible path to system verification.

Chapter 6

Correct Code Generation

CAmkES' RPC connectors provide a mechanism for one component instance to seamlessly invoke a function provided by another component instance. This mechanism works by introducing generated code into both instances. The correctness of a component instance implicitly relies on this colocated generated code.

This chapter discusses a technique for proving the correctness of generated C code for one of CAmkES' RPC connectors, seL4RPCSimple. Recall that, as outlined in Section 2.4.3, the correctness property we wish to show is that a remote function invocation is equivalent to a local function invocation. The remote function call involves parameter marshalling, an IPC transfer, parameter unmarshalling and then calling the user-provided implementation. This is followed by the reverse process of marshalling, transferring and unmarshalling return values. The equivalence we aim to show is depicted informally in Figure 6.1.

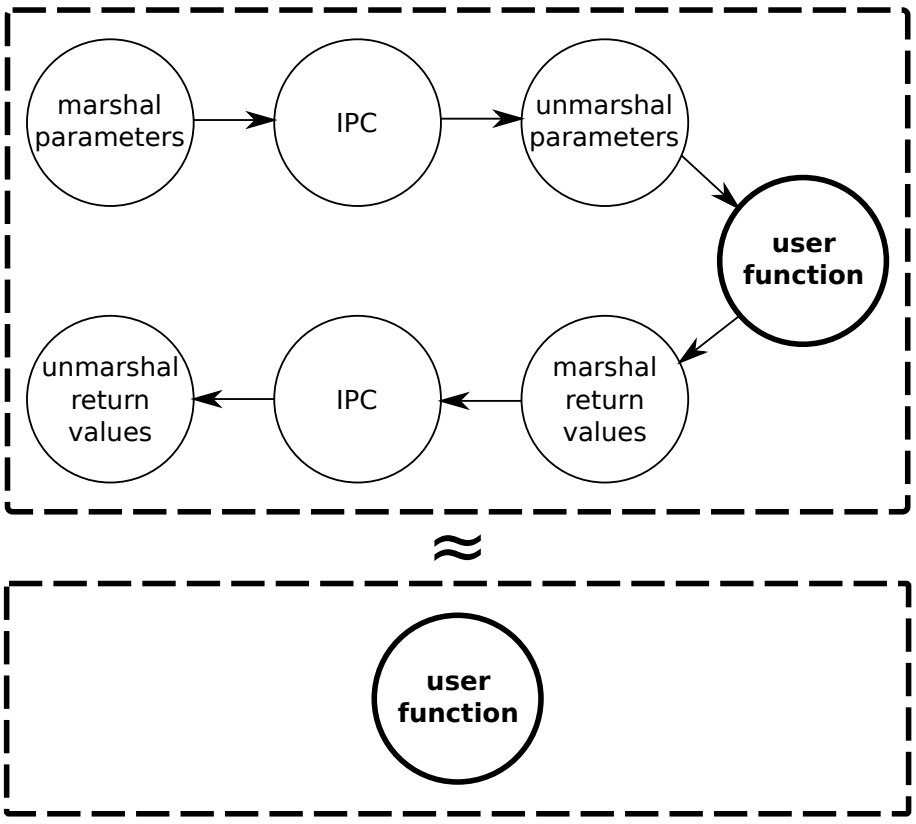

Figure 6.1: Remote and local function invocation equivalence

Our approach leverages previous work on manual, interactive verification of C code, but shows how to automate the proofs of sufficiently regular code. The generated proofs can be manually connected to hand written proofs about user code that interoperates with the generated code, forming a correctness guarantee of the composed system equivalent to a fully hand written proof. Through this process, we achieve code-level functional correctness with a high degree of expressivity and assurance. In comparison to the formalisms of Chapter 4 and Chapter 5, we sacrifice a degree of proof automation to achieve a significantly reduced semantic gap between the model and implementation. Though the implementation presented is specific to C code, the underlying approach is applicable to other languages, with the proviso that most other languages additionally depend on a runtime that would be implicitly trusted. Sections of this work have been the subject of a previous publication (Fernandez, Andronick, et al. 2015a) and technical report (Fernandez, Andronick, et al. 2015b).

6.1 Utility

A static component platform like CAmkES operates by generating communication stubs – glue code – prior to compilation time. This glue code is then compiled with hand written, user-provided code to form the full program for a component instance. The control flow within a component instance interleaves glue code with hand written code and the correctness of the system depends on both. A bug in the glue code can undermine concerted efforts to guarantee correctness of hand written code.

As an example of incorrect glue code, consider generated code for an RPC connection where the callee's glue code unmarshals parameters in the reverse order to that in which the caller's glue code marshalled them. This error is similar to reading and writing data with conflicting endianness, and can lead to problems which are difficult to debug. In part, the debugging complexity stems from the assumed correctness of glue code. Just as a programmer's intuition is not to suspect a compiler fault as the most likely cause of a bug, the user of a component platform is conditioned not to suspect faulty code generation by the platform.

While it could be argued that the logic error described above could be exposed by testing, it is not feasible to exhaustively test glue code. There are an infinite number of possible interface definitions, each of which produces unique glue code. Furthermore, there are classes of glue code bugs that can never be caught by testing. To quantify this, we describe a genuine bug that was present in CAmkES' glue code that was only discovered during verification. The bug can be observed in a simple system. Consider two component instances communicating over an RPC connection, as depicted in Figure 6.2.

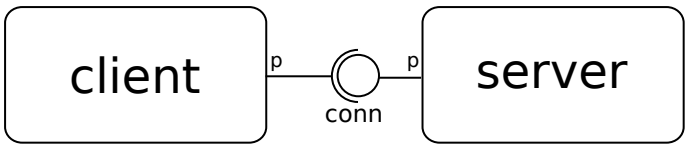

Figure 6.2: 64-bit echo server

The two component instances, `client` and `server`, use a simple procedure containing a single method for echoing a 64-bit signed integer. The entire ADL specification for the system is as follows.

```
procedure P {
  int64_t echo(in int64_t x);
}

component Client {
  control;
  uses P p;
}

component Server {
  provides P p;
}

assembly {
  composition {
    component Client client;
    component Server server;

    connection seL4RPCSimple conn(from client.p, to server.p);
  }
}
```

This system was built and run successfully. With reference to the intended role of glue code explained in Chapter 2, the glue code for the system appeared as expected and manual audits of this code – including one by the current author – had not raised any issues.

When `client` calls `p_echo`, glue code marshals the input parameter into the process' IPC buffer. An abbreviated and commented version of the glue code that was generated is as follows.

```
/* Hardware word size (bits) */
#define __WORDSIZE 32

/* Write `mr` to the word at offset `i` in the IPC buffer */
extern void seL4_SetMR(int i, seL4_Word mr);

unsigned int p_echo_marshal(int64_t x) {
  unsigned int length = 0;

  /* Marshal the method index */
  seL4_SetMR(length, 0);
  length++;

  /* Marshal the input */
  seL4_SetMR(length, (seL4_Word)x);
  length++;
  seL4_SetMR(length, (seL4_Word)(x >> __WORDSIZE));
  length++;

  return length;
}
```

On the callee's side, the glue code generated for `server` unmarshals this input from the process' IPC buffer. This corresponding code appeared as follows.

```
1 /* Read the word at offset `i` in the IPC buffer */
2 extern seL4_Word seL4_GetMR(int i);
3
4 void p_echo_unmarshal(int64_t *x) {
5   unsigned int mr = 1; /* Skip over 0 that contained the method index */
6
7   /* Unmarshal the input */
8   *x = ((int64_t)seL4_GetMR(mr)) | (((int64_t)seL4_GetMR(mr + 1)) << __WORDSIZE);
9 }
```

When verifying the above code, it was impossible to discharge one of the proof obligations. This remaining goal pointed to a subtle bug. An experienced practitioner may wish to try and spot the problem themselves before continuing to the next paragraph.

The template for the callee's glue code, from which the above was generated, type cast the result of `seL4_GetMR` to the type of the parameter being unmarshalled. In most cases this was the correct action. However, in the above glue code, a negative input value of `x` causes undefined behaviour in the callee's glue code according to the C standard. To see this, consider the steps by which `x` is transmitted:

1. The low 32 bits of `x` are written into the caller's IPC buffer;
2. The high 32 bits of `x` are shifted down and written into the caller's IPC buffer;
3. The low 32 bits of `x` are read out of the callee's IPC buffer; then
4. The high 32 bits of `x` are read out of the callee's IPC buffer and shifted up.

It is this final shift that is undefined behaviour when `x` has a negative value. The relevant clause from the C99 standard (ISO/IEC 2005) is §6.5.7.4:

> *The result of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. If E1 has an unsigned type, the value of the result is E1 * 2^{E2}, reduced modulo one more than the maximum value representable in the result type. If E1 has a signed type and nonnegative value, and E1 * 2^{E2} is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.*

In two's complement representation, the highest bit of a number is used to indicate its sign; 0 for positive, 1 for negative. Thus a negative value of `x` would result in the third word of the IPC buffer containing a 32-bit value with the 31st bit set. After casting this to a signed 64-bit type, shifting it left by 32 bits results in a value not representable in a signed 64-bit type, violating the last sentence of the above rule.

This problem is demonstrated for the input value, −42, in Figure 6.3. The representation of −42 in two's complement form is given in the uppermost row of bits. The right shift and down casts to store the value in two words of the IPC buffer are correct.[1] Similarly, reading the two words out of the IPC buffer and up casting these values to signed 64-bit values are also defined operations. However, the left shift of the bits destined for high word of the result is not. A single undefined subexpression causes the containing expression and the code as a whole to have undefined behaviour.

[1] How the vacated bits of a signed type are filled when a negative value is right shifted is implementation defined, according to §6.5.7.5 of the C99 standard. Here we have depicted those bits as being 1-filled. The glue code's correctness does not depend on the fill mask for this shift.

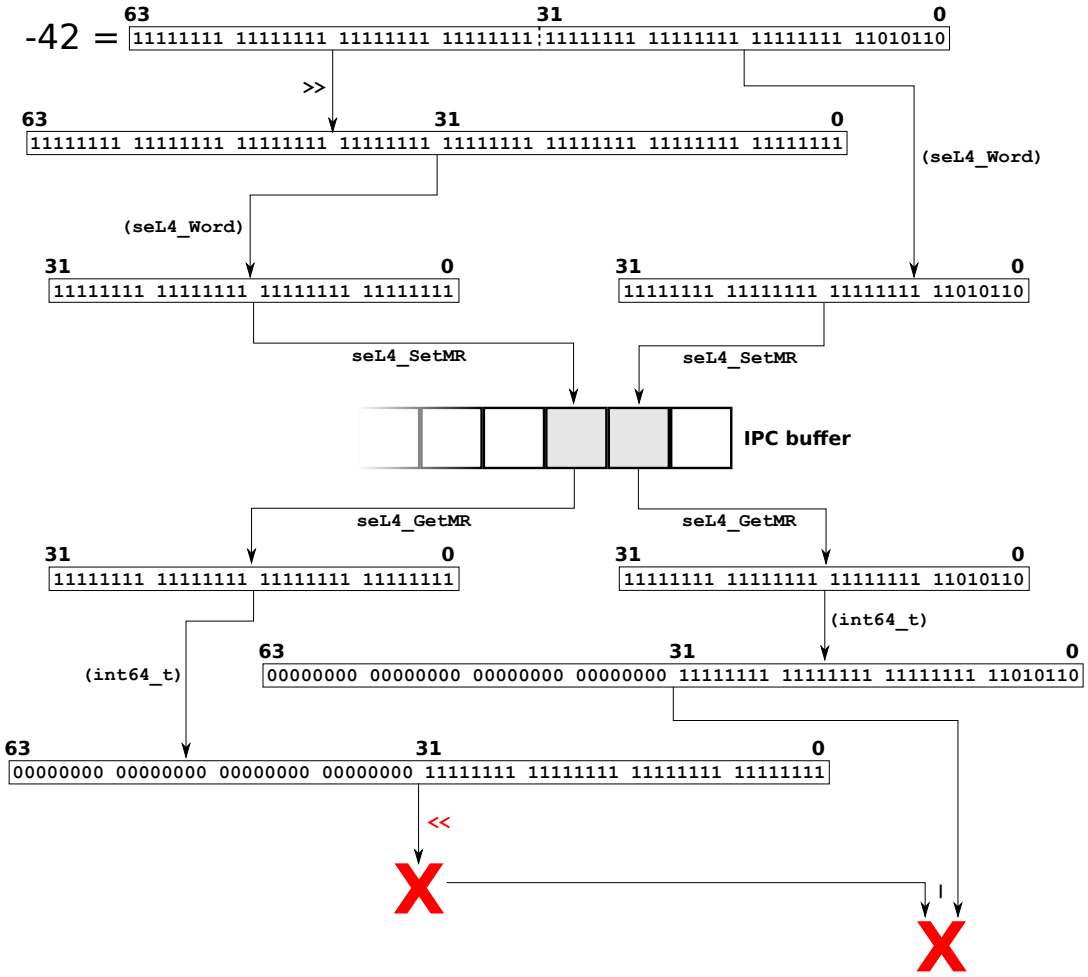

Figure 6.3: Marshalling and unmarshalling of a signed 64-bit value involving undefined behaviour

The fix for this issue was to perform the shift on an unsigned type where this rule is not violated:[2]

```
1  void p_echo_unmarshal(int64_t *x) {
2    unsigned int mr = 1; /* Skip over 0 that contained the method index */
3
4    /* Unmarshal the input */
5    *x = (int64_t)(((uint64_t)seL4_GetMR(mr)) |
6      (((uint64_t)seL4_GetMR(mr + 1)) << __WORDSIZE));
7  }
```

This bug was discovered during verification of the offending glue code and is unlikely to have been found otherwise. Glue code is generally not inspected by users – who assume its correctness – and our C compiler in use at the time emitted instructions that always produced the expected result, even with negative input. However, another conformant C compiler would be within its remit to make optimisations that relied on the value of x being positive.

The bug we have just described is subtle, but not unusual in the context of serialisation code in low level languages. The C standard deliberately specifies numerous expressions as having undefined behaviour, many

[2]This points to a more general wisdom that is not the subject of this thesis, but nevertheless a valuable intuition for C programmers. In general, unsigned types are safer and more straightforward to reason about than signed types. This is largely driven by the numerous cases of undefined behaviour in the C99 standard that apply to signed types, but not to unsigned types (for example, integer overflow). The interested reader is referred to (Dietz et al. 2012; Greenaway 2015; Norrish 1998; Wang et al. 2012) for more details.

of which are related to edge cases in the type system. These instances are usually poorly understood by programmers and often intimately related to typical operations performed by serialisation code (Dietz et al. 2012; Formaggio 2015; Wang et al. 2013).

The generated proofs we detail in this chapter guarantee not only the absence of undefined behaviour in the glue code, but also give the glue code a precise semantics. The produced proofs prevent the possibility of logic errors, like that given at the beginning of this section, from passing unnoticed and free the system designer from concerning themselves with the glue code's correctness. When building or debugging a component-based system, the designer now need only ensure the correctness of their hand written code.

6.2 Proof Generation

Our goal is to provide a collaborative proof environment where the proofs of generated code are, themselves, generated and integrate seamlessly with hand written proofs. That is, we wish to enable the designer to provide a pervasive system correctness proof as if they had manually proven the correctness of the RPC glue code, without requiring this laborious process. To achieve this, we take a translation validation approach, as introduced in Section 3.4.

Alongside the RPC glue code, the CAmkES code generator produces a specification for each piece of glue code and a proof of conformance to this specification. These specifications are given in the form of Hoare triples in the style shown in Section 2.3.2. By using the same idioms and structures in the generated specifications as a hand written formalism would use, reuse of the generated lemmas is made intuitive for the designer.

The generated specifications are precise, in that they describe the exact preconditions required for correctness and the exact postcondition capturing the effect of the glue code. This is unusual for code-level proofs in an interactive theorem prover. In general, when manually proving the correctness of a function, the specification of that function and its callees are made only as precise as necessary to prove the property of interest. This pattern is often extended to providing multiple underspecifications of a function when proving multiple correctness properties; one per property. The motivation for this is to reduce proof effort to a minimum. It is often significantly more complicated, or even intractable, to prove a precise specification for a function.

By the same reasoning, underspecification would ease the effort required in constructing our proof generation process, but we avoid this because we do not know what property the user will want to prove. Underspecifying a glue code function can result in the user being unable to prove their property of interest and having to fall back on manually reasoning about glue code. To prevent this undesirable situation, we choose to give a complete and precise specification at the expense of a more complicated proof generator.

There is a mutual dependency between glue code and user-provided code, similar to that between processes in a Rely/Guarantee framework (Jones 1983; Owicki and Gries 1976). Both pieces of code have conditions they need the other to respect in order to guarantee their own behaviour. The proofs we are generating provide a guarantee of the glue code – both what it does and what it does not do – and they require well-behaved user-provided code in order to uphold this guarantee. In particular, the glue code has a set of invariants the user-provided code must maintain and some private state that the user-provided code must not read or write. This will be explained in more concrete terms in the following section.

6.3 Example

To demonstrate our proof-producing process, we work through verifying an example CAmkES system. Its specification is as follows.

```
1  procedure Swapper {
2    unsigned int swap(inout int a, inout int b);
3  }
4
5  component Client {
6    control;
7    uses Swapper cs;
8  }
9
10 component Service {
11   provides Swapper ss;
12 }
13
14 assembly {
15   composition {
16     component Client c;
17     component Service s;
18
19     connection seL4RPCSimple conn(from c.cs, to s.ss);
20   }
21 }
```

6.3.1 User-provided and Generated Code

This system has two component instances, `c` that needs to exchange two integers, and `s` that provides a service for doing such. Recalling Section 2.2.1, observe that this will involve two pieces of user-provided code and two pieces of glue code generated by CAmkES. The interaction between these artefacts is depicted in Figure 6.4.

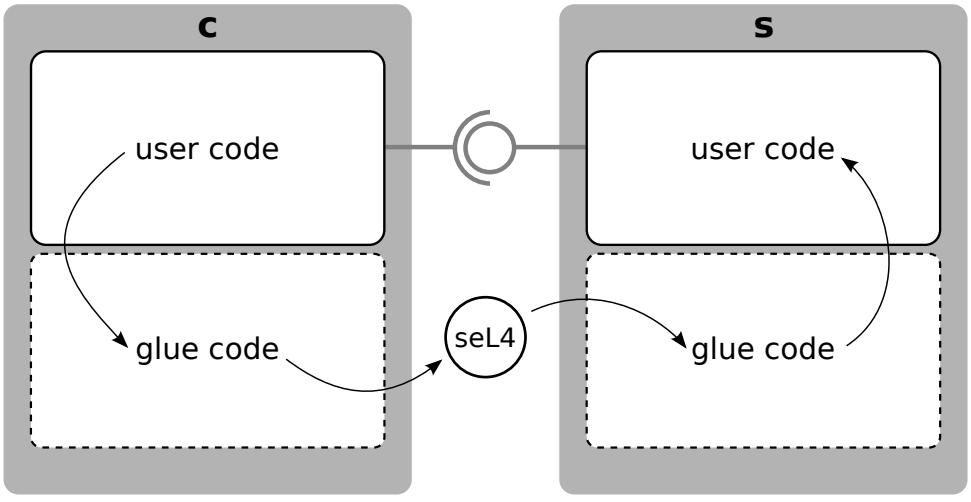

Figure 6.4: Control flow between user-provided and generated code

To understand precisely how this control flow occurs, it is informative to examine the code of this system. The user-provided code of `Client` demonstrates how `c` initiates this exchange of integers.

```
int run(void) {
  int x = 3;
  int y = 5;

  unsigned int i = cs_swap(&x, &y);

  return 0;
}
```

Conversely, the user-provided source of `Server` shows how the swapping of these two integers is achieved. While swapping the two integers, it also returns a count of how many swap operations have been performed.

```
static unsigned int counter;

unsigned int ss_swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;

    counter++;

    return counter;
}
```

Obviously this example is quite artificial, but it has been chosen to demonstrate that our process can deal with bidirectional parameters, pointers and global persistent state.

The natural correctness property of this system – that the values pointed to by two pointers are correctly swapped by `ss_swap` – is straightforward to come up with in isolation from any consideration of the glue code. We can formulate this in Isabelle/HOL and the proof of a local function invocation is straightforward.

```
  "∀s0. {|λs. s = s0 ∧ ptr_valid_s32 s x ∧ ptr_valid_s32 s y|}
          ss_swap x y
        {|λr s. r = counter s0 + 1 ∧ counter s = r ∧
                heap_w32 s (ptr_coerce x) = heap_w32 s0 (ptr_coerce y) ∧
                heap_w32 s (ptr_coerce y) = heap_w32 s0 (ptr_coerce x)|}!"
```

The precondition requires that `x` and `y` are valid pointers. More precisely, the expression `ptr_valid_s32 st p` requires that the pointer `p` is a valid reference to a signed 32-bit value in state `st`. The postcondition states that the value of the global `counter` will be updated and returned and that the values at the pointers `x` and `y` will be swapped. The function `ptr_coerce` is analogous to type casting in C and the expression `heap_w32 st p` returns the value pointed to by `p` in the state `st`. Thus this lemma states that, given two valid pointers to signed 32-bit integers, the function `ss_swap` exchanges their pointed-to values, increments `counter` and returns its new value. Note that, in this lemma, we refer to inputs, outputs and the global state. We do not constrain the expressivity the system designer has in their precondition and postcondition.

If `cs_swap` represented a direct call to `ss_swap`, a composed proof would be possible at this point. However, recall that `cs_swap` is actually a call into glue code and, similarly, the call to `ss_swap` comes from glue code. So to achieve a composed proof we need formal specifications for both pieces of glue code. To have context for the generated proofs that will follow, we first look at the glue code itself. The generated code for the caller, `c`, follows the same templated format from Section 6.1, but deals with both marshalling and unmarshalling.

```c
static unsigned int cs_swap_marshal(int a, int b) {
  unsigned int index = 0;

  /* Marshal the method index */
  seL4_SetMR(index, 0);
  index++;

  /* Marshal the first input */
  seL4_SetMR(index, (seL4_Word)a);
  index++;

  /* Marshal the second input */
  seL4_SetMR(index, (seL4_Word)b);
  index++;

  return index;
}

static void cs_swap_call(unsigned int length) {
  seL4_MessageInfo_t info = seL4_MessageInfo_new(0, 0, 0, length);

  /* Call the seL4 endpoint */
  (void)seL4_Call(6, info);
}

static unsigned int cs_swap_unmarshal(int *a, int *b) {
  unsigned int index = 0;

  /* Unmarshal the return value */
  unsigned int ret = (unsigned int)seL4_GetMR(index);
  index++;

  /* Unmarshal the first output */
  *a = (int)seL4_GetMR(index);
  index++;

  /* Unmarshal the second output */
  *b = (int)seL4_GetMR(index);
  index++;

  return ret;
}

unsigned int cs_swap(int *a, int *b) {
  unsigned int length = cs_swap_marshal(*a, *b);

  cs_swap_call(length);

  unsigned int ret = cs_swap_unmarshal(a, b);

  return ret;
}
```

Similarly, the glue code for the callee, `s`, deals with unmarshalling method inputs and then marshalling outputs.

```
/* User-provided implementation */
extern unsigned int ss_swap(int *a, int *b);

/* Per-thread globals */
static int swap_a_1;
static int swap_a_2;
static int swap_b_1;
static int swap_b_2;

static int *get_swap_a(void) {
  switch (camkes_get_tls()->thread_index) {
    case 1:
      return &swap_a_1;
    case 2:
      return &swap_a_2;
    default:
      assert(!"invalid thread index");
      abort();
  }
}

static int *get_swap_b(void) {
  switch (camkes_get_tls()->thread_index) {
    case 1:
      return &swap_b_1;
    case 2:
      return &swap_b_2;
    default:
      assert(!"invalid thread index");
      abort();
  }
}

static void ss_swap_unmarshal(int *a, int *b) {
  unsigned int index = 1;

  /* Unmarshal the first input */
  *a = seL4_GetMR(index);
  index++;

  /* Unmarshal the second input */
  *b = seL4_GetMR(index);
  index++;
}

static unsigned int ss_swap_invoke(int *a, int *b) {
  /* Call the user's implementation */
  return ss_swap(a, b);
}

static unsigned int ss_swap_marshal(unsigned int ret, int a, int b) {
  unsigned int index = 0;

  /* Marshal the return value */
  seL4_SetMR(index, (seL4_Word)ret);
```

```
56    index++;
57
58    /* Marshal the first output */
59    seL4_SetMR(index, (seL4_Word)a);
60    index++;
61
62    /* Marshal the second output */
63    seL4_SetMR(index, (seL4_Word)b);
64    index++;
65
66    return index;
67  }
68
69  unsigned int ss_swap_internal(void) {
70    int *a = get_swap_a();
71    int *b = get_swap_b();
72
73    ss_swap_unmarshal(a, b);
74
75    unsigned int ret = ss_swap_invoke(a, b);
76
77    unsigned int length = ss_swap_marshal(ret, *a, *b);
78
79    return length;
80  }
```

Note that we omit the top-level entry point for this glue code, that is an event loop, and instead consider `ss_swap_internal`, that is called repeatedly by the event loop, to be the outermost function. This is driven by limitations in the control flow that can be expressed in the framework on which we build. Firstly, it is difficult to state properties about functions that do not terminate, such as an event loop. Secondly, it is difficult to capture the semantics of blocking system calls whose scope to affect global state is beyond what is visible in C. That is, the postcondition of a system call is the effect of the kernel's execution. We discuss this in more depth in Section 9.1, but for the purposes of the current proof, we effectively implicitly axiomatise the execution of `seL4_Wait`/`seL4_ReplyWait` in this glue code.

Recall that the control flow in the system at runtime consists of an interleaving of the user-provided and generated code. Extending Figure 6.4 with the code of this system gives us Figure 6.5.

6.3.2 Proof Generation

The correctness property we wish to prove was introduced in Section 2.4.3. With reference to this specific example, we wish to prove that a sequence of operations performed by the interleaving of glue code and user code is equivalent to directly calling `ss_swap`. The sequence of operations we define are as follows.

```
1  "swap a b ≡ do cs_swap_marshal ( *a) ( *b);
2                 ss_swap_internal;
3                 cs_swap_unmarshal a b
4              od"
```

These steps are: marshalling in `c` (line 1), then the entire execution in `s` (line 2), including unmarshalling, executing `ss_swap`, and marshalling results, and finally unmarshalling of the result in `c` again (line 3). It can be observed that this sequence matches the control flow of the component system at runtime.

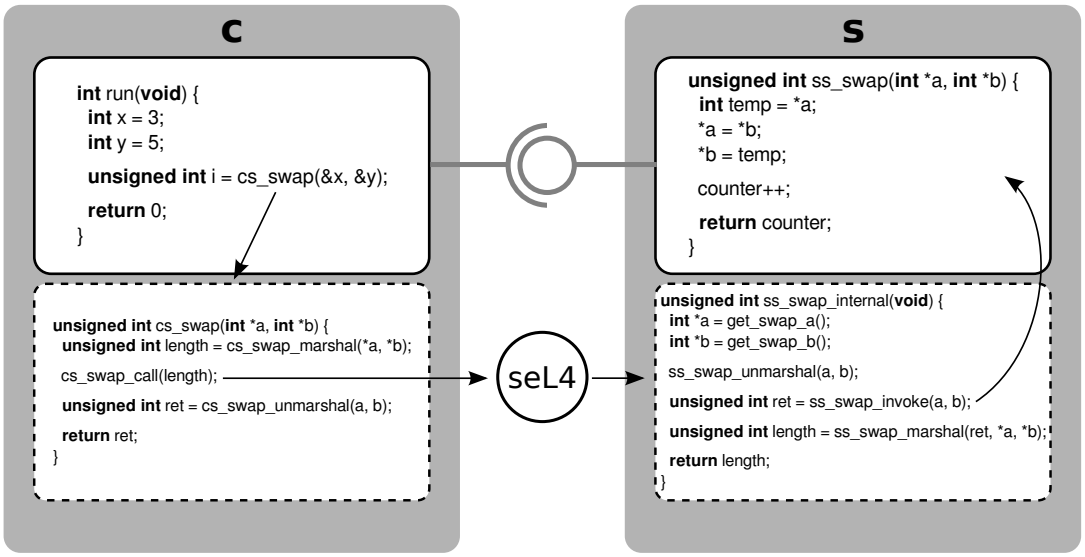

Figure 6.5: Control flow between user-provided code and glue code

It may not be immediately obvious, but the precondition and postcondition that our generated proof needs to show must be parameters of the theorem. This is because we do not know in advance what property the system designer wishes to prove. Stated another way, the proof generation process has no awareness of the correctness criteria of the system. In fact, it is an explicit goal of ours to enable the designer to state correctness criteria *after* these proofs have been generated.

To parameterise these proofs, we generate them inside a *locale*. Recall that Appendix B contains a more expansive introduction to locales. For the purposes of our present discussion, the relevant aspect of locales is that they allow us to specify the precondition and postcondition as typed parameters. By providing a precondition, postcondition and proof that their function satisfies these *locally*, a system designer is delivered a proof that their precondition and postcondition are satisfied by a *remote* invocation of their function. Our fundamental aim with this mechanism is to give the designer RPC proofs with the equivalent manual effort of local proofs.

Our toolchain for translating C into Isabelle/HOL (see Section 2.3.1 and Section 2.3.2) already constructs a locale containing the function representations it produces, so we extend this to add our own parameters and assumptions. We introduce the generated locale definition piece-by-piece. The first block, shown below, fixes the parameters for the user-provided precondition and postcondition. Note that the precondition accepts a `lifted_globals` state parameter, in order to be able to reference the values of globals, and the initial value of each of the function parameters. Similarly the postcondition accepts two state parameters to be able to talk about the values of globals before and after execution. It accepts five further parameters: the return value, the values of the two function parameters before execution and their values after execution (recall that they are both `inout` parameters). In this example, `conn_pruned` is the locale originating from AutoCorres' abstraction of the glue code functions.

```
1  locale "conn_pruned_glue" = "conn_pruned" +
2
3    fixes swap_P :: "lifted_globals ⇒ 32 sword ⇒ 32 sword ⇒ bool"
4    fixes swap_Q :: "lifted_globals ⇒ lifted_globals ⇒ 32 word ⇒ 32 sword ⇒
5                     32 sword ⇒ 32 sword ⇒ 32 sword ⇒ bool"
```

The next block contains the assumption that the precondition and postcondition hold on the user's implementation. This may be surprising to see, but it should be intuitively clear that a generated proof of the

user-provided precondition and postcondition holding over an RPC invocation would not be possible if the precondition and postcondition did not actually hold over a local invocation.

```
1   assumes ss_swap_wp':"∀s0 p0_in p1_in.
2
3     {|λs. s = s0 ∧ inv s ∧
4          ptr_valid_s32 s p0 ∧ p0_in = ucast (heap_w32 s (ptr_coerce p0)) ∧
5          (p0 = Ptr (symbol_table ''swap_a_1'') ∨
6           p0 = Ptr (symbol_table ''swap_a_2'')) ∧
7          ptr_valid_s32 s p1 ∧ p1_in = ucast (heap_w32 s (ptr_coerce p1)) ∧
8          (p1 = Ptr (symbol_table ''swap_b_1'') ∨
9           p1 = Ptr (symbol_table ''swap_b_2'')) ∧
10         swap_P s p0_in p1_in|}
11
12       ss_swap p0 p1
13
14    {|λr s. inv s ∧
15         (∃p0_out p1_out. p0_out = ucast (heap_w32 s (ptr_coerce p0)) ∧
16          p1_out = ucast (heap_w32 s (ptr_coerce p1)) ∧
17          swap_Q s0 s r p0_in p0_out p1_in p1_out)|}!"
```

The form of this assumption is the familiar correctness Hoare triple given previously, but there are some additions. The predicate `inv` (lines 3 and 14) captures certain invariants that the glue code requires. This ensures that the user's function does not disrupt any state private to the glue code. This is necessary in a language like C with no enforced scoping of memory accesses. The precondition and postcondition conjuncts referencing `p0_in`, `p1_in`, `p0_out` and `p1_out` (lines 4, 7, 15 and 16) provide us with variables referring to the values of the pointers at either end of execution. The precondition also contains conjuncts specifying that the pointers passed to the user's `ss_swap` are confined to a set of private glue code globals (lines 5, 6, 8 and 9). It is unlikely the user would need these assumptions directly, but they can be used to justify that the input pointers do not alias any other data `ss_swap` relies upon. Finally we have the precondition and postcondition that the user themselves provide, `swap_P` (line 10) and `swap_Q` (line 17), respectively.

The next block requires the user-provided code to respect the private state of the glue code, as introduced in Section 6.2. It refers to a specification, `setMR`, that describes modifying a word of the IPC buffer. In a CAmkES system, the IPC buffer is considered part of the implementation and hence private to the glue code. These assumptions capture the requirement that neither the user's precondition nor their postcondition can rely on the contents of the IPC buffer. Any wellformed precondition and postcondition will trivially satisfy these assumptions.

```
1   assumes swap_pre_stable_setmr[simp]:
2     "⋀s i x. swap_P (setMR s i x) = swap_P s"
3
4   assumes swap_post_stable_setmr1[simp]:
5     "⋀ s i x. swap_Q (setMR s i x) = swap_Q s"
6
7   assumes swap_post_stable_setmr2[simp]:
8     "⋀ s s' i x. swap_Q s (setMR s' i x) = swap_Q s s'"
```

Similar to the requirement on the IPC buffer, the next block captures the requirement that the precondition and postcondition cannot depend on the values of the global variables used internally by the glue code. Once again, these are trivially satisfied by any wellformed precondition and postcondition.

```
  assumes swap_pre_stable_update_a[simp]:
    "⋀v s p. ⟦p = Ptr (symbol_table ''swap_a_1'') ∨
              p = Ptr (symbol_table ''swap_a_2'')⟧
      ⟹ swap_P (update_s32 s p v) = swap_P s"

  assumes swap_post_stable_update_a[simp]:
    "⋀v s p. ⟦p = Ptr (symbol_table ''swap_a_1'') ∨
              p = Ptr (symbol_table ''swap_a_2'')⟧
      ⟹ swap_Q (update_s32 s p v) = swap_Q s"

  assumes swap_pre_stable_update_b[simp]:
    "⋀v s p. ⟦p = Ptr (symbol_table ''swap_b_1'') ∨
              p = Ptr (symbol_table ''swap_b_2'')⟧
      ⟹ swap_P (update_s32 s p v) = swap_P s"

  assumes swap_post_stable_update_b[simp]:
    "⋀v s p. ⟦p = Ptr (symbol_table ''swap_b_1'') ∨
              p = Ptr (symbol_table ''swap_b_2'')⟧
      ⟹ swap_Q (update_s32 s p v) = swap_Q s"
```

The next assumption requires that the execution of the user's code does not modify the valid pointers in the system. Note that this pertains to modifying the collection of pointers *themselves*, not their *values*. This requirement is driven by the type system of the framework we are working within and no legitimate C function provided by the user should ever violate this assumption.

```
  assumes ptr_valid_s32_stable_swap:
    "⋀s s' r p0_in p0_out p1_in p1_out. swap_Q s s' r p0_in p0_out p1_in p1_out
      ⟹ ptr_valid_s32 s' = ptr_valid_s32 s"
```

The final assumption requires that the globals used by the glue code have distinct addresses. This is always the case for a correctly loaded C program.

```
  assumes dist[simp]:
    "distinct [symbol_table ''swap_a_1'', symbol_table ''swap_a_2'',
               symbol_table ''swap_b_1'', symbol_table ''swap_b_2'']"
```

With these assumptions and locale fixes in place, we generate a series of fine-grained lemmas proving complete specifications for the glue code on a function-by-function basis. The reason it is possible to generate the proofs along with the statement of these lemmas is that we have knowledge of the logic that generates the glue code. By being able to anticipate the form of a function, we can also anticipate the proof steps required to prove its specification. We elide these individual proofs, but present the final correctness lemma that is proven by chaining the preceding lemmas together. Note that it is once again made possible by exploiting our knowledge of how this sequence of functions will appear. The lemma demonstrates that the user's precondition and postcondition, `swap_P` and `swap_Q` respectively, are satisfied by the sequence of glue code operations that are equivalent to a local function call.

```
1  lemma cs_ss_swap_equiv_wp: "∀s0.
2
3    {λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s p0_out ∧ ptr_valid_s32 s p1_out ∧
4         distinct [((ptr_coerce p0_out)::32 word ptr),
5                   ((ptr_coerce p1_out)::32 word ptr)] ∧
6         (∀s1 s2 v. swap_Q s1 (update_s32 s2 p0_out v) = swap_Q s1 s2) ∧
7         (∀s1 s2 v. swap_Q s1 (update_s32 s2 p1_out v) = swap_Q s1 s2) ∧
8         swap_P s p0 p1}
9
10     do cs_swap_marshal p0 p1;
11        ss_swap_internal;
12        cs_swap_unmarshal p0_out p1_out
13     od
14
15   {λr s. inv s ∧ swap_Q s0 s r p0 (ucast (heap_w32 s (ptr_coerce p0_out)))
16                           p1 (ucast (heap_w32 s (ptr_coerce p1_out)))}!"
```

While the lemma appears more dense than the previously outlined Hoare triple, it is not significantly more complicated. The `inv s` condition (lines 3 and 15) is familiar from the previous term, as are the `ptr_valid_s32` conditions (line 3). A new condition we have incurred is that the two pointers involved in the swap operation are not equal (lines 4-5). This is not strictly required for correctness, but the generated proofs conservatively require the absence of aliasing between any user-provided pointers. This limitation could be lifted relatively easily, but it would require a way for the user to indicate whether pointer aliasing was permissible in their callee implementation. That is, if we were to naively remove the requirement for distinct pointers and the user's callee implementation relied on pointer distinctness, the generated lemma would no longer be strong enough to be of use to them.

The two conjuncts in the precondition that involve the user's postcondition, `swap_Q` (lines 6 and 7), may also be unexpected. These require that the user's postcondition must not access the values of the pointer arguments through its final state parameter. This seems counterintuitive, but we use it to allow internal glue code variables to substitute for the user's pointers when marshalling and unmarshalling arguments. That is, the user's postcondition can depend on the *values* of the arguments, but cannot depend on their *addresses*. The final conjunct (line 8) states that the user's precondition must hold prior to execution.

The postcondition is simpler, merely stating that the user's postcondition holds in addition to the glue code invariants (lines 15 and 16). Here, `ucast` converts an unsigned 32-bit value to a signed 32-bit value.

To reinforce what we have shown, both the lemma and its proof are generated fully automatically. The proof relies on further generated lemmas of glue code functions, that are accomplished by exploiting the regularity of the glue code. With knowledge of the logic that generates the glue code, we can predict its structure and formulate the template of a proof. The proof generation process has a natural congruence with code generation in a component platform, and we have designed CAmkES' proof generation to reuse its code generation engine.

6.3.3 User Instantiation

With a generated proof in hand, it remains for the user to instantiate the locale with their specific precondition and postcondition and to discharge the locale assumptions. The natural precondition for the swap function is simply validity of the pointers, which is already subsumed by the generated precondition. Thus the user can instantiate their precondition to the trivial `λ s a b. True`.

The natural postcondition needs to capture the property that the values have been swapped, and also that the counter has been incremented (recall that the counter is a global variable). We provide this as follows, but note

that we also need to claim that the validity of pointers is preserved; that is, `ss_swap` does not invalidate any `int` pointers. This condition is relied upon by the generated proofs in assuming that internal pointers used for marshalling and unmarshalling are not invalidated by running user code. Observe that, in this postcondition, we are able to state properties of both inputs, outputs and the state.

```
"λs0 s r a a_out b b_out. r = counter s0 + 1 ∧ counter s = r ∧
                         is_valid_w32 s0 = is_valid_w32 s ∧
                         a_out = b ∧ b_out = a"
```

Discharging the assumptions of the locale is mostly trivial in this case, as it generally would be for any well-behaved user code. In general, any user code should not touch the glue code's internal state, and by keeping to this guideline the only potentially challenging assumption to discharge is that the precondition and post-condition provided actually hold over the user's function. Note that this is the same proof obligation the user would be facing in a system with only their local functions and no component platform. A slight complication is introduced by having to also provide a symbol table, a mapping from strings to addresses. This is an artefact of the current implementation and can be instantiated with an artificial mapping for the purposes of our proofs. Having proven the assumptions, an instantiated variant of the previous lemma is produced.

```
lemma instantiated_final_verbose: "∀s0.

  {λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s p0_out ∧ ptr_valid_s32 s p1_out ∧
       distinct [p0_out, p1_out] ∧
       (∀s1 s2 v. (λr a a_out b b_out. r = counter s1 + 1 ∧
                     counter (update_s32 s2 p0_out v) = r ∧
                     is_valid_w32 s1 = is_valid_w32 (update_s32 s2 p0_out v) ∧
                     a_out = b ∧ b_out = a) =
                  (λr a a_out b b_out. r = counter s1 + 1 ∧
                     counter s2 = r ∧
                     is_valid_w32 s1 = is_valid_w32 s2 ∧
                     a_out = b ∧ b_out = a)) ∧
       (∀s1 s2 v. (λr a a_out b b_out. r = counter s1 + 1 ∧
                     counter (update_s32 s2 p1_out v) = r ∧
                     is_valid_w32 s1 = is_valid_w32 (update_s32 s2 p1_out v) ∧
                     a_out = b ∧ b_out = a) =
                  (λr a a_out b b_out. r = counter s1 + 1 ∧
                     counter s2 = r ∧
                     is_valid_w32 s1 = is_valid_w32 s2 ∧
                     a_out = b ∧ b_out = a)) ∧
        True}

    do cs_swap_marshal p0 p1;
       ss_swap_internal;
       cs_swap_unmarshal p0_out p1_out
    od

  {λr s. inv s ∧ r = counter s0 + 1 ∧ counter s = r ∧
          is_valid_w32 s0 = is_valid_w32 s ∧
          ucast (heap_w32 s (ptr_coerce p0_out)) = p1 ∧
          ucast (heap_w32 s (ptr_coerce p1_out)) = p0}!"
```

This looks quite dense, but observe that many of the conjuncts are now trivial and can be elided. Applying Isabelle/HOL's automated simplification yields the following, more concise formulation. Note that this is much simpler than the original, generic variant.

```
lemma instantiated_final_simplified: "∀s0.
  {|λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s p0_out ∧ ptr_valid_s32 s p1_out ∧
        distinct [p0_out, p1_out]|}
    do cs_swap_marshal p0 p1;
       ss_swap_internal;
       cs_swap_unmarshal p0_out p1_out
    od
  {|λr s. inv s ∧ r = counter s0 + 1 ∧ counter s = r ∧
          is_valid_w32 s0 = is_valid_w32 s ∧
          ucast (heap_w32 s (ptr_coerce p0_out)) = p1 ∧
          ucast (heap_w32 s (ptr_coerce p1_out)) = p0|}!"
```

This lemma can now be reused in further hand written proofs. To demonstrate this, we prove that swapping the same pointers twice returns them to their original values. The proof of the following is achieved in fifteen straightforward steps, by utilising the instantiated version of `cs_ss_swap_equiv_wp`.

```
lemma swap_twice: "∀s0.
   {|λs. s = s0 ∧ inv s ∧ ptr_valid_s32 s x ∧ ptr_valid_s32 s y ∧ x ≠ y|}
     do swap x y;
        swap y x
     od
   {|λr s. inv s ∧ r = counter s0 + 2 ∧ counter s = r ∧
      ucast (heap_w32 s (ptr_coerce y)) = ucast (heap_w32 s0 (ptr_coerce y)) ∧
      ucast (heap_w32 s (ptr_coerce x)) = ucast (heap_w32 s0 (ptr_coerce x))|}!"
```

This final lemma shows a correctness property on code that contains interleaved calls to user code and glue code, and eventual invocation of a remote function. Observe that the effort required on the part of the user has been only manually proving the correctness of the code they have written, their implementation of `ss_swap`. We have used an example containing a single function for simplicity, but this process is capable of handling an arbitrary number of methods in an procedure interface, and scales linearly in the number of methods. The automated process we have shown has reduced the manual steps required in proving an RPC invocation to simply the manual steps for proving a local invocation.

6.4 Limitations and Future Directions

The proof generation process we have described is paired with a specific CAmkES connector, seL4RPCSimple, that handles the primitive CAmkES data types (`character`, `int`, `uint8_t`, etc), but not `string`, arrays or user-defined types (C structs). Dealing with these compound types would most likely require extensions in order to marshal and unmarshal runtime-dependent data formats. That is, marshalling and unmarshalling would need to perform `strcpy`- or `memcpy`-like operations, whose termination argument depends on values that are not constant at generation-time. This possibility is expanded upon in Section 9.2.

The semantics of the seL4 system calls used by glue code is currently axiomatised. This leaves a vulnerable gap with the potential for semantic mismatch between the formalisms of seL4 and CAmkES. A separate line of ongoing work is aimed at giving a formal semantics for seL4's system call stubs. With this established, it would be possible to formally connect the generated proofs of this chapter with the semantics of the system call stubs, giving a top-to-bottom guarantee of the correctness of the glue code operating on top of seL4. This is extrapolated further in Section 9.1.

Given the assumptions we require of user-provided code, it is natural to consider the semantics of a system wherein code deliberately violates these assumptions. To give some idea of what is possible, malicious user-provided code could directly access the underlying endpoint capability and send malformed data. While the

implementation of the seL4RPCSimple connector attempts to be robust against accidental or malicious misuse of its glue code, the proofs we have demonstrated only provide a semantics for well-behaved use. That is, in the presence of malformed invocations of glue code, reasoning will be reduced to a fully manual proof of generated code. It does not make sense to attempt to prove the "correctness" of an incorrect use of glue code, but it may be valuable to prove that a trusted recipient of malformed messages always detects and handles the error safely. This would be a valuable future extension to our work.

6.5 Other Connectors

The generated proofs we have demonstrated describe only how to achieve correctness for an RPC connector. These proofs are not applicable to CAmkES' other two modes of communication, events and dataports. Our focus on RPC has been motivated by the complexity of the RPC glue code. The logic for code generation of RPC is far more extensive than either events or dataports and presents a greater risk for potential bugs. The same proof strategy provides less benefit for events and dataports as these mechanisms are predominantly implemented by the seL4 kernel itself, rather than CAmkES. We expand on this with respect to each communication mode below.

6.5.1 Code Generation for Events

Emitting and consuming CAmkES events is implemented on seL4 by sending and receiving over an asynchronous endpoint. The CAmkES primitives naturally correspond to the seL4 system calls for operating on an asynchronous endpoint. To demonstrate this clearly, we consider the following simple example system.

```
component Sender {
  emits Ev e;
}

component Receiver {
  consumes Ev e;
}

assembly {
  composition {
    component Sender sender;
    component Receiver receiver;

    connection seL4AsynchNative conn(from sender.e, to receiver.e);
  }
}
```

Code generation produces the following piece of glue code for `sender`.

```
void e_emit(void) {
  seL4_Notify(5, /* ignored */ 0);
}
```

The glue code for `receiver` is somewhat longer, but not sufficiently more complicated.

```
1  /* Badge indicating that a message was pending */
2  #define PENDING 195894762
3
4  int e_poll(void) {
5    seL4_Word badge = seL4_Poll(6);
6    return badge == PENDING;
7  }
8
9  void e_wait(void) {
10   seL4_Wait(6, NULL);
11 }
```

It is clear from these listings that the glue code functions, `e_emit`, `e_wait` and `e_poll`, are simply shallow wrappers around the seL4 API. A generated proof of their correctness in the style of those shown previously in this chapter would consist of nothing substantial beyond an axiomatisation of the corresponding system call's semantics and then a trivial proof by unfolding. The value of such a proof without connection to a formal semantics of the seL4 system calls is negligible, which is why we have not pursued this avenue.

6.5.2 Code Generation for Dataports

Recalling Section 2.2.1, dataports are CAmkES' abstraction of shared memory. Where the code generation for events was minimal, the code generation for dataports is essentially nil. The glue code for both component instances with a common dataport is a single global with the name and size of the dataport. In this scenario a proof of "correct code" has no useful meaning. The real correctness criteria for dataports is that the two regions of memory in separate address spaces map to the same region of physical memory at runtime. The most promising way to achieve a guarantee of this is through the formalism introduced in Chapter 7, but we quantify in Section 9.3 why we have left this for future work.

6.6 Summary

This chapter has demonstrated a novel technique that improves the state of the art in component platform verification. By providing generated proofs for generated code, we both increase the assurance of the CAmkES component platform and also decrease the manual effort required in guaranteeing whole system correctness properties. The process removes pre-existing assumptions on the correctness of CAmkES glue code. The generated correctness proofs free the system designer from having to trust the RPC glue code, and thus allow them to focus their efforts on other key parts of their system.

Chapter 7

Correct Architecture Implementation

The process of starting a CAmkES system is analogous to booting a conventional operating system, complete with a number of potential pitfalls and attack vectors. The correctness of a component architecture and component code can be easily undermined by an initialisation process that fails to ensure that the system at runtime is a faithful representation of the artefacts that are visible at design time. A correctness guarantee of the initialisation process cements the relationship between what is verified and what is ultimately executed, providing proof that the component boundaries are in place and cannot change at runtime. This chapter details CAmkES' initialisation mechanism, the possible problems that can occur and the formal guarantees we provide that these problems do not manifest. We go on to generate an access control policy and demonstrate how to reason about information flow properties using this policy.

7.1 Initialisation Process

The required initial configuration of a CAmkES system is completely described by the architectural description. That is, the architecture of a given CAmkES system provides sufficient information in isolation to determine how the system should appear in its initial state. The required kernel objects and capabilities to them can be inferred directly from the architecture, with the exception of virtual address space dimensions and memory contents. Leveraging this property, CAmkES systems are initialised with a generic seL4 root task (described in Section 2.2.1.6) that is parameterised with an initialisation specification derived directly from the architecture description. This initialisation specification is provided to the root task by compiling a representation of the specification into its data section at build time.

After initialising the hardware, seL4 transfers control to this root task. The task iterates over its provided specification, creating kernel objects and capabilities as necessary. The task is single threaded and, on completion, suspends itself. It also runs at seL4's highest priority level, a level that is unavailable to CAmkES component instances. The task never provides its subordinates with capabilities to either its CSpace or its TCB. This scenario allows us to guarantee, by design, that the task runs uninterrupted until it is complete and then permanently removes itself from the kernel's run queue. From this point onwards, though its objects and capabilities still exist, they are effectively inert and cannot influence the running system as they are unreachable from the remaining operational threads.

The state of the system after the root task has suspended itself is intended to correspond to the architecture description. Because all authority in seL4 is conferred by capabilities, any connection between two compo-

nent instances in a CAmkES specification results in a corresponding capability path[1] between those instances, either direct or indirect. For the standard seL4 connectors, this results in *shared objects*, that is objects to which more than one component instance has a capability. In particular:

- a synchronous endpoint shared between two component instances connected by a seL4RPCSimple connection;
- an asynchronous endpoint shared between two component instances connected by a seL4AsynchNative connection; and
- page capabilities to the same frame installed into the page tables of two component instances connected by a seL4SharedData connection.

We aim to show that these are the only connections that exist between components and that they match the ADL specification of the system. The initialisation specification that describes these kernel objects and capabilities is generated at build time from the architecture description by the CAmkES code generator. The correctness of the generic root task is the subject of other work (Boyton 2014; Boyton et al. 2013) and is out of scope for the present discussion. However, a faithful translation of the architecture description into an initialisation specification is a guarantee that the work in this chapter attempts to establish, by providing the system designer with an environment to verify that the expected properties of their architecture hold. The next section examines the potential problems that can occur in the architecture translation and thus what the formalism of this chapter is guarding against.

7.2 Possible Problems

To guarantee that a running system matches a pre-defined architecture description, it is essential to understand the ways in which it could differ if the initialisation process was incorrect. It is also relevant to understand the potential sources of this incorrectness. In this discussion, we only consider faults stemming from the CAmkES code generator, as faults in the seL4 kernel or the root task are out of scope as outlined previously and addressed in previous work.

Consider a system that contains three component instances: `a`, `b` and `c`. In this system, `a` has an RPC connection to `b` and an event connection to `c`.

```
1  assembly {
2    composition {
3      component A a;
4      component B b;
5      component C c;
6
7      connection seL4RPCSimple conn1(from a.rpc, to b.rpc);
8      connection seL4AsynchNative conn2(from a.ev, to c.ev);
9    }
10 }
```

This system is depicted in Figure 7.1.

Given this description, the initialisation specification should contain three CSpaces, one for each component instance. Component instance `a` should have capabilities to a synchronous endpoint and an asynchronous

[1]The kernel objects and capabilities to them form a directed graph. Furthermore, influence of one object over another is observable in this graph as a path from one to the other. For a more extensive discussion of capability graphs, the reader is referred to the seL4 reference manual (Trustworthy Systems Team 2014a).

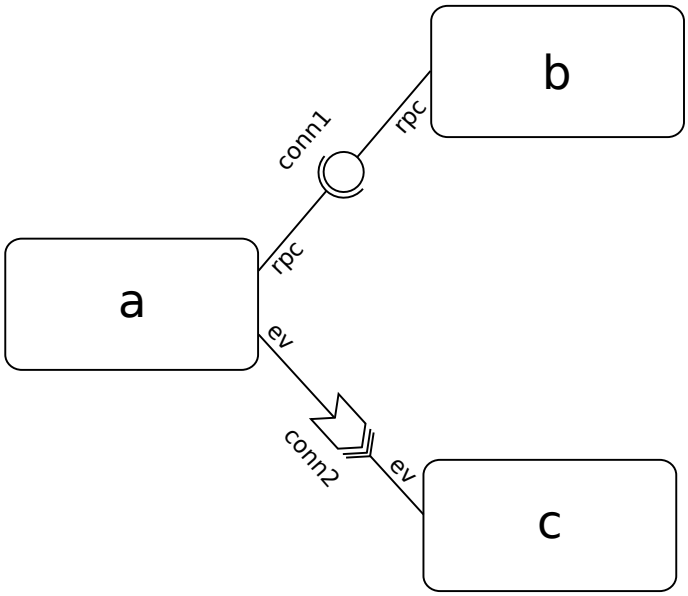

Figure 7.1: A system of three component instances

endpoint, corresponding to its interfaces `rpc` and `ev`, respectively. Component instance `b` should have a capability to this same synchronous endpoint and component instance `c` should have a capability to the asynchronous endpoint.

This configuration has some relevant security properties that the system designer may wish to leverage. The connector, seL4AsynchNative, provides one way information flow; that is, the sender cannot learn anything from the receiver. In the system we've described, this means the execution of `c` should not be able to affect the execution of `a`. By extension, `c` should also be unable to affect the execution of `b`.

Now suppose the generation of this initialisation specification was incorrect and it granted an extra capability to the synchronous endpoint to `c`. This variation is depicted in Figure 7.2. This would result in an unintended information leakage from `c` to the other two component instances. While the extra capability is observable in the initialisation specification, it may be difficult to notice in a more complicated system. Moreover, because the initialisation specification is an intermediate build artefact to the system designer, it may not be manually inspected.

All problems with the initialisation specification manifest as inaccurate capability graphs, but there are more subtle examples than the one just discussed. For example, a capability to one of the code pages of component instance `a` may be incorrectly mapped into one of the page tables of component instance `c`, allowing `c` to tamper with `a`'s code at runtime, again effecting an information leakage.[2]

While there are more complicated examples than this, it suffices to say that all possible incorrect translations are observable as incorrect implied authority within the capability graph. This is a specific property that makes a capability-based microkernel like seL4 a well-suited target for component platform proofs. One category of unfaithful translation that is not detected by the work described in this chapter is missing capabilities or objects. Such a situation can only ever lead to more constrained authority, and is thus not a security violation. Missing capabilities or objects are detectable as unexpectedly absent runtime functionality, and can easily be caught by testing.

[2] Recall that paging structures in seL4 are completely described by kernel objects and capabilities to them. That is, a page mapping is represented as a capability to a frame object that is hosted within a page table object. Hence, this incorrect mapping is still observable in the initialisation specification.

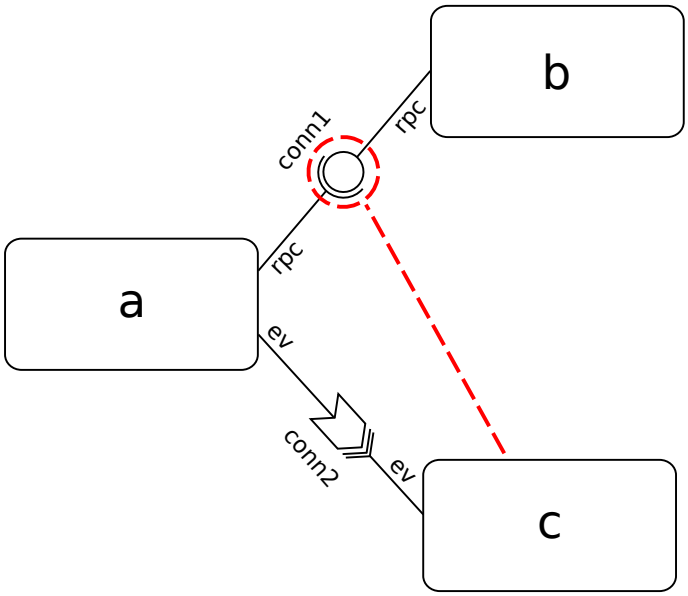

Figure 7.2: A system with an incorrect extra capability

7.3 Approach

Section 2.2.1.6 described how a system designer's architecture is interpreted into a running system by CAmkES. The CapDL specification produced by CAmkES can be translated into a series of Isabelle/HOL definitions by an existing tool (Lewis et al. 2014). This automatic translation gives a formalised representation of the initial system state. Previous work gives a semantics to such definitions and provides a framework for reasoning about information flow and authority confinement in a seL4-based system (Kuz et al. 2010; Murray et al. 2012; Sewell et al. 2011). The maturity and expressivity of this framework make it an ideal environment for reasoning about information flow within a CAmkES system. The system designer can use this to validate that their architecture has the safety and security properties they expect. Thus our task becomes closing the gap between the designer's original architecture description and the translated CapDL specification.

We take a hybrid approach, building on ideas from translation validation (Pnueli, Siegel, and Singerman 1998), in verifying the correspondence of the two artefacts: a CAmkES architecture and a CapDL specification. We represent the former of these in the formalism described in Chapter 4. This model gives a precise definition of the architecture and is also produced automatically by the CAmkES code generator. Firstly we have manually written an Isabelle/HOL definition of CAmkES' CapDL-generation logic. This is a function that takes a CAmkES architecture and auxiliary information as input and produces a CapDL specification and an access control policy as output. We expand on the precise type definition of this function in the next section.

To guarantee that the output of the Isabelle/HOL function we have written is consistent with the CapDL specification that the implementation produces, we have modified the CAmkES code generator to produce another output, an Isabelle/HOL theory containing generated lemmas. These lemmas prove the equivalence of the output of our Isabelle/HOL function with the concrete output of the implementation.

The advantage of, and primary motivation for, this design is scalability and expressiveness for the designer. The concrete specification is a large Isabelle/HOL term that is cumbersome to reason about. However, properties that generalise can be proven on a universally quantified output of the generator function. To this end, we have proven that the specification produced by the generator always refines the access control policy pro-

duced. For a system designer, this means they are spared of proving this complex property on their large, concrete specification and can simply instantiate this general lemma. We describe the specifics of how this can be achieved later in this chapter.

7.4 CapDL Generator Function

Our Isabelle/HOL function that generates a CapDL specification and access control policy takes three inputs: a CAmkES architecture, a collection of address space objects and a collection of unassigned interrupt nodes. The latter two may be surprising, and they are only necessary due to implementation details.

The need for an auxiliary collection of address space objects is driven by the way in which CAmkES defines a component instance's virtual address space. Rather than manually specifying address space regions or attempting to predict a component instance's requirements, CAmkES allows the compiler and linker full control over address space layout. This gives the compiler much greater scope for optimisation and instrumentation.[3] CAmkES then derives the frames, page tables and page directory required by a component instance post hoc by inspecting its compiler-produced binary. Naturally, we do not wish to encode the address space layout algorithm of a full compiler in our generator function, so we instead phrase it as accepting an input that fully describes the frames, page tables and page directories. As a consequence, there are some necessary conditions on this auxiliary collection for it to be wellformed, that we will discuss later in this chapter.

The collection of unassigned interrupt nodes is motivated by a much more mundane reason. CAmkES infers an interrupt node for every hardware interrupt defined by the target platform. These can be assigned to an asynchronous endpoint for delivery of the interrupt if the system makes use of them. However, for the purposes of this verification effort we limit ourselves to systems that do not use interrupts and hence we require all the inferred interrupt nodes to be unassigned. This is not a fundamental limitation, but merely a proof convenience that could be lifted in future. All objects in a CapDL specification have a numeric identifier, represented as a 32-bit word. Their identifiers are incremented linearly as CAmkES defines objects, with the interrupt nodes defined last. This means that the identifiers of the interrupt nodes follow the address space objects, that are defined by the second input parameter. While it would be possible to have the generator function derive the interrupt node identifiers given only its first two inputs, we simplify it by requiring a third input that completely describes these interrupt nodes. Again, this necessitates conditions for this collection to be wellformed, but the requirements are significantly simpler than for the collection of address space objects.

The output of the generator function is an option type of a pair of CapDL specification and access control policy. The function produces `None` if one of its inputs is not wellformed. Conversely, given a wellformed CAmkES architecture with accurately described address space objects and interrupt nodes that fits our assumptions, this function always produces a CapDL specification and an access control policy. The specification, as discussed previously, completely describes the kernel objects and capabilities to them in the initial state of the system. This initial state is actually persistent, as the limitations of CAmkES make it immutable at runtime by design.[4] The policy describes a series of information domain labels (one per component instance and one per connection) and an upper bound on the effect any label can have on any other label.

[3] This may seem like a trivial motivation, but modern embedded C code typically relies heavily on the compiler's ability to perform optimisations like dead code elimination and function inlining. Some systems rely on more esoteric optimisations like GCC's `-ffunction-sections`. By not attempting to control the address space layout of a component instance, CAmkES can effectively "stay out of the compiler's way" and give it the greatest chance of producing efficient code.

[4] Though it may seem like a complex property, it is actually trivial to observe that a CAmkES-derived CapDL specification is immutable. Any modification to a capability or the essential state of a kernel object must happen by operating on a containing CNode capability, by virtue of the seL4 API design. No CNode capabilities are addressable by CAmkES component instances. This is an intentional outcome motivated by our target use case of *static* component-based systems. For more extensive justification of these claims, the reader is referred to the proofs of separation kernel bisimilarity in the seL4 proofs (Trustworthy Systems Team 2014b).

In full, the type signature of the generator function appears as follows.

```
1  extended_generate :: "camkes_state ⇒ cdl_heap ⇒ irqs
2                        ⇒ (cdl_state × label PAS set) option"
```

In this context, `camkes_state` is a type synonym for `assembly` from Chapter 4, `cdl_heap` is a mapping from CapDL object identifiers to objects and `irqs` is a record type capturing the collection of interrupt nodes. The output types `cdl_state` and `PAS` are the types of a CapDL specification and an access control policy, respectively. We expand on the `PAS` type later in this chapter.

7.5 Assumptions

In defining the Isabelle/HOL function that represents CAmkES' CapDL generation logic, we have made several assumptions that are encoded into the conditions for the inputs being wellformed. That is, the predicates that encode these checks contain some properties that should be true of all correctly described CAmkES systems and some properties that constrain the systems to which this formalism can be applied. The constraints are, in part, driven by the necessary conditions for a CapDL specification to refine an access control policy, formalised in the relation `pcs_refined` that we will return to later. Further details on the requirements for this refinement are available in previous work (Klein et al. 2014).

In this section, we give an overview of the restrictions we place on input artefacts by dealing with each of the validity predicates in turn. When reading the following, it is important to note that all inputs to these predicates are generated. That is, these predicates never need to be referenced manually by the system designer and they are automatically proven to hold over a valid CAmkES specification.

7.5.1 Validity of CAmkES Specification

The first property we require is that the CAmkES specification itself is valid.

```
1  definition
2    valid_spec :: "camkes_state ⇒ bool"
3  where
4    "valid_spec spec ≡ ({garbage, irq_label, idle_label} ∩
5           set (map fst (components (composition spec)) @
6                  map fst (connections (composition spec))) = {}) ∧
7             length (components (composition spec)) < unat maxDomain ∧
8               wellformed_assembly spec ∧
9                 (∀(inst, comp) ∈ set (components (composition spec)).
10                   ∀i ≤ length (provides comp) + length (consumes comp).
11                     ∃f. ipc_buffer inst i = Some f) ∧
12   (∀(inst, _) ∈ set (components (composition spec)). mangle_pd inst ∈ dom id_of)"
```

There are three special information domain labels, `garbage`, `irq_label` and `idle_label`. These three labels are applied to numeric identifiers that map to, respectively, no valid object, unassigned interrupt nodes and the idle thread.[5]. Lines 4-6 capture that no component instance name or connection name can collide

[5]The "idle thread" is, conceptually, the action that is taken when the kernel scheduler's run queue is empty; that is, when no threads are schedulable. In most operating systems, the idle thread is an abstract concept, not directly visible in the implementation. In contrast, seL4 represents the idle thread as an explicit, schedulable entity. An advantage of this approach is that confidentiality and integrity proofs can explicitly describe and restrict communication channels involving the scheduler (Murray et al. 2012; Murray et al. 2013). A side effect is that the idle thread is visible in CapDL specifications and must be explicitly dealt with. This is all automatic and transparent to the designer working on top of our formalism, as we assume no component system's correctness relies on interacting with the idle thread.

with these. This is purely for convenience, as component instance names and connection names are used as labels. The `garbage`, `irq_label` and `idle_label` labels are entirely arbitrary and can be set to any value, providing they themselves are distinct.

Line 7 specifies that the number of component instances in the system must be less than the number of scheduler domains.[6] This is because each component instance runs in a separate domain, with one domain reserved for the idle thread. The predicate from Chapter 4 for a CAmkES specification to be wellformed appears on line 8. Lines 9-11 refer to an automatically generated function, `ipc_buffer`. This function takes a component instance name and an index as inputs, and returns the frame hosting the IPC buffer of the thread of the given index in the given component instance. This conjunct states that each thread in the system must have an IPC buffer.[7] Finally, the last conjunct on line 12 requires that the name of each component instance's page directory must be associated with a valid object.

7.5.2 Validity of Address Space Objects' Collection

The auxiliary collection of address space objects provided to the generator function as its second input has three separate predicates that describe what it means for it to be valid. These three are separated, rather than defined as a single predicate, due to definition and locale scoping within the contained theory. The amalgamated validity requirement can be thought of as the conjunction of these three. The first two take as their first parameter an initial collection of objects that is produced by the generator *prior* to appending the address space objects. The third takes the system architecture specification. The reason for these additional parameters becomes clear when enumerating the conjuncts that make up these predicates.

First, we discuss the predicate, `valid_extra`. The collection of address space objects is provided as a mapping from object identifier, `cdl_object_id`, to CapDL object, `cdl_object`. This type is abbreviated to `cdl_heap`, as seen with reference to the address space objects parameter, `extra`.

```
1  definition
2    valid_extra :: "cdl_state ⇒ cdl_heap ⇒ bool"
3  where
4    "valid_extra initial extra ≡
5       (∀obj ∈ ran extra. isPageDirectory obj ∨ isPageTable obj ∨ isFrame obj) ∧
6       dom (cdl_objects initial) ∩ dom extra = {} ∧
7       (∀obj ∈ ran extra. case obj of
8            PageDirectory pd ⇒ (∀cap ∈ ran (cdl_page_directory_caps pd).
9              case cap of
10               PageTableCap i _ None ⇒
11                 isSomePageTable (extra i) ∧ one_cap extra i
12             | FrameCap i _ _ _ None ⇒
13                 isSomeFrame (extra i) ∧ one_cap extra i
14             | _ ⇒ False)
15         | PageTable pt ⇒ (∀cap ∈ ran (cdl_page_table_caps pt).
16             case cap of
17               FrameCap i _ _ _ None ⇒
18                 isSomeFrame (extra i) ∧ one_cap extra i
19             | _ ⇒ False)
20         | _ ⇒ True) ∧
```

[6] Note that the scheduler domains referred to here are the mechanism by which seL4 isolates threads in the manner of a separation kernel. This concept of domains is distinct from the information domains we discuss, though CAmkES systems are configured with one scheduler domain per information domain, so in practice they are always one-to-one for the purposes of this work.

[7] It might seem peculiar to have such an assumption, but it is possible to configure a thread on seL4 that does not have an IPC buffer. Such a thread can run indefinitely, provided it does not attempt system calls or interprocess communication. In CAmkES, we disallow this as the glue code assumes the existence of a valid IPC buffer. Regardless, it should be clear that a thread without an IPC buffer is of limited value as it can perform computation but no I/O.

```
21      (∀obj_id ∈ dom extra. case extra obj_id of
22           Some (PageTable _) ⇒ one_cap extra obj_id
23         | Some (Frame _) ⇒ one_cap extra obj_id
24         | _ ⇒ True) ∧
25      (∀obj_id ∈ dom extra. obj_id ≠ idle_thread_ptr) ∧
26      inj (extra_filter_out_frames extra)"
```

This definition uses the functions `dom` and `ran` that extract the domain and range of a function, respectively. Injectivity of a function is expressed by the predicate `inj`. Line 5 captures that all objects in this collection must be either page directories, page tables or frames, which is to be expected. Line 6 expresses that none of the object identifiers of these address space objects should collide with any object identifiers already used by the generator function. The next conjunct that spans lines 7-20 appears formidable, but actually captures a very simple property; namely, that each contained capability references an object of consistent type and is the only capability to that object in the collection. This property becomes necessary later when expressing the disjoint nature of address spaces. Lines 21-24 state that there is a single capability to each page table and frame. That is, there are no orphaned page tables or frames in this collection. The penultimate conjunct on line 25 states that no address space object has the identifier of the idle thread. The final conjunct on line 26 states that removing the frames from this collection yields a resulting injective function. Again, this is necessary later for expressing the disjointness of address spaces.

The second validity predicate, with identical type signature, contains a separate series of conjuncts.

```
1  definition
2    valid_extra' :: "cdl_state ⇒ cdl_heap ⇒ bool"
3  where
4    "valid_extra' initial extra ≡
5      (∀n i f. ipc_buffer n i = Some f ⟶ f ∈ dom extra ∧
6       (∃frame. extra f = Some (Frame frame)) ∧
7        label_frame extra f = Some n) ∧
8      (∀obj_id ∈ dom extra. case extra obj_id of
9          Some (PageDirectory _) ⇒
10           ∃name. id_of (mangle_pd name) = Some obj_id
11       | _ ⇒ True)"
```

The first conjunct on lines 5-7 states that the frames backing the IPC buffers of all threads are in this collection, are indeed frames and are labelled with the information domain label of the component instance to which they relate. The second on lines 8-11 states that each page directory in the collection is the designated page directory of one of the component instances.

The final predicate claims two properties of the address space objects.

```
1  definition
2    valid_extra'' :: "camkes_state ⇒ cdl_heap ⇒ bool"
3  where
4    "valid_extra'' spec extra ≡
5      (∀(name, _) ∈ set (components (composition spec)).
6        ∃pd. extra (the_id_of (mangle_pd name)) = Some (PageDirectory pd)) ∧
7      (∀obj_id∈dom extra.
8        case extra obj_id of
9            None ⇒ True
10         | Some (cdl_object.PageDirectory x) ⇒
11             ∃name ∈ fst ` set (components (composition spec)).
12               id_of (mangle_pd name) = Some obj_id
13         | Some _ ⇒ True)"
```

Firstly, it states that for each component instance in the system, the collection must contain a corresponding page directory (lines 5-6). Secondly, it states that each page directory in the collection has a corresponding component instance (lines 7-13). Taken together, they form a requirement for a one-to-one correspondence between component instances and page directories.

There is some redundancy in these three predicates. However, thus far we have not attempted to prune superfluous clauses or optimise expressions because the predicates are only ever used in generated proofs. Their verbosity does not impede the system designer because they are never unfolded in hand written proofs.

Note that the majority of the conjuncts we have detailed for these three predicates are not constraints of the systems we support, but rather type system constraints. The type definitions of `cdl_heap` and `cdl_object` are not strong enough to prevent the definition of terms that are internally inconsistent. This is driven by the need for mutually recursive references in the foundation of these types (objects contain capabilities, which in turn refer to objects), and is not accidental.

The only emergent constraint from these three predicates that represents a limitation on the category of systems we can handle is that capabilities to the same frame cannot appear in two page tables.[8] By implication, this formalism does not support systems that involve shared memory (dataports in CAmkES parlance). This is not a fundamental limitation, and the decision to make this compromise was partially motivated by proof convenience. However, lifting this restriction requires more substantial considerations that we elaborate on in Section 9.3.

7.5.3 Validity of Interrupt Nodes' Collection

In contrast to the CAmkES specification and address space objects, defining the concept of validity for the collection of interrupt nodes is simpler. The reason for this is that there is very little variability in this collection. We require all the nodes to be unassigned and the platform we target has a fixed number of interrupts, 256. As noted in Section 7.3, this collection is sufficiently regular that it could be derived from the preceding two inputs, to avoid supplying it entirely. However, there would be little value in making this modification as the generator function's inputs and its invocation are all automatic, so the existing structure does not impede the system designer.

Building on the previous two inputs, the predicate for validity of the interrupt nodes' collection takes the initial collection of objects produced by the generator prior to appending the address space objects and the collection of address space objects as its first two parameters.

```
1  definition
2    valid_irqs :: "cdl_state ⇒ cdl_heap ⇒ irqs ⇒ bool"
3  where
4    "valid_irqs initial extra irqs ≡
5       range (irqs_map irqs) = dom (irqs_objects irqs) ∧
6       dom (cdl_objects initial) ∩ dom (irqs_objects irqs) = {} ∧
7       dom extra ∩ dom (irqs_objects irqs) = {} ∧
8       (∀x ∈ ran (irqs_objects irqs). case x of
9          CNode c ⇒ c = (|cdl_cnode_caps = empty, cdl_cnode_size_bits = 0|)
10       | _ ⇒ False) ∧
11      idle_thread_ptr ∉ range (irqs_map irqs)"
```

The first property on line 5 states an internal consistency property of `irqs`, that it must reference the interrupt nodes that it itself defines. Lines 6 and 7 state that the identifiers of the interrupt nodes must not collide with

[8] More precisely, a capability to the same frame cannot appear in any two places; two page tables, two page directories, or a page table and a page directory. Recall that, on ARM, the largest page sizes are mapped directly into page directories.

the identifiers in use by the initial objects produced by the generator or the address space objects. Lines 8-10 state that every interrupt node must be empty (unassigned). Finally, line 11 states that no interrupt node must have the identifier of the idle thread.

The only constraint within this predicate that represents a system limitation – that is, not a type system consistency property – is that the interrupt nodes must all be unassigned. This prevents use of this formalism with systems that make use of hardware interrupts. This is purely a simplification made to ease proofs in this experimental prototype. This limitation could be removed with relatively low effort, but by retaining it we enable the possibility of using previous information flow work that contains this assumption (Murray et al. 2012). This is not a restriction of other seL4 security results (Sewell et al. 2011), so it may be desirable to make this constraint optional in future.

7.6 CapDL Correspondence

Section 7.4 described an Isabelle/HOL function for producing a CapDL specification of a component-based system. However, a system is initialised by a specification that is produced by the CAmkES code generator. To have confidence that properties proven on the output of the Isabelle/HOL function are applicable to the running system, we need a guarantee that the output matches the specification produced by the code generator.

As in previous chapters, we take an approach of translation validation. Firstly, an Isabelle/HOL representation of the code generator's output is produced by an existing tool (Lewis et al. 2014). Secondly, we have modified CAmkES' code generator to produce another Isabelle/HOL theory proving equality between the two concrete artefacts: its own CapDL specification output and the output of the generator function when operating on the current system's architecture description.

The lemmas proven in this generated theory are straightforward. They first demonstrate that the generated specification satisfies the assumptions of the generator function (`valid_spec`, `valid_extra`, etc.). Their proofs operate via automated tactics and, where unavoidable, exhaustive enumeration. It is possible for a system designer to prove these lemmas manually, but this involves numerous tedious steps. The final lemma guarantees the generator function produces a result (not `None`) and its output corresponds to the code generator's output, here named `final`.

```
1  lemma final:"∃pases. extended_generate spec extra irqs = Some (final, pases)"
```

The second output of the generator function, `pases`, is the subject of the following section.

7.7 Policy Refinement

The definitions we have described in this chapter formalise the logic by which CAmkES generates a CapDL specification. However, it is the second output of the generator function, the access control policy, that is of most interest to the system designer. It is the policy that provides an intuitive definition of the upper bound of information flow within a CAmkES system. One of its fields contains a set of information domain labels (one per component instance and one per connection) and labelled, directed edges between them. Together, these form an authority graph.

The type of this policy is formalised in an Isabelle/HOL record, `PAS` (policy, abstraction, subject).

```
1  record 'a PAS =
2    pasObjectAbs :: "'a agent_map"
3    pasASIDAbs :: "'a agent_asid_map"
4    pasIRQAbs :: "'a agent_irq_map"
5    pasPolicy :: "'a auth_graph"
6    pasSubject :: "'a"
7    pasMayActivate :: "bool"
8    pasMayEditReadyQueues :: "bool"
9    pasMaySendIrqs :: "bool"
10   pasDomainAbs :: "domain ⇒ 'a"
```

This record is parameterised by a type, `'a`, for the labels in the system. The four fields, `pasObjectAbs`, `pasASIDAbs`, `pasIRQAbs` and `pasDomainAbs`, assign labels to the CapDL objects, address space identifiers, interrupts and scheduler domains, respectively. The `pasPolicy` field contains the aforementioned authority graph. The `pasSubject` field indicates the currently active label (the label of the currently running thread). Finally, the fields `pasMayActivate`, `pasMayEditReadyQueues` and `pasMaySendIrqs` describe the abilities of the currently active label. This record captures the full set of information required for reasoning about confidentiality and integrity. A deeper explanation of this type and associated mechanisms is available in prior work (Klein et al. 2014; Sewell et al. 2011).

The policy we produce is a set of this record type.

```
1  definition
2    pas_of :: "camkes_state ⇒ cdl_heap ⇒ irqs ⇒ label PAS set"
3  where
4    "pas_of spec extra irqs ≡ {pas.
5       pasObjectAbs pas = poa_of spec extra irqs ∧
6       (∀asid. pasASIDAbs pas asid = garbage) ∧
7       (∀irq. pasIRQAbs pas irq = irq_label) ∧
8       pasPolicy pas = policy_of spec extra irqs ∧
9       pasSubject pas ∈ {idle_label} ∪ fst ` set (components (composition spec)) ∧
10      ¬ pasMayActivate pas ∧
11      ¬ pasMayEditReadyQueues pas ∧
12      ¬ pasMaySendIrqs pas ∧
13      pasDomainAbs pas = domain_of spec
14    }"
```

It may be surprising that we produce a set, rather than a single, definitive policy. Our motivation is that we wish to allow variance in certain fields of the record. For example, we permit the active label (the `pasSubject` field) to be any label associated with a component instance or the idle thread (line 9). The authority graph (the `pasPolicy` field) is defined in full as follows.

```
definition
  policy_of :: "camkes_state ⇒ cdl_heap ⇒ irqs ⇒ label auth_graph"
where
  "policy_of spec extra irqs ≡

     (* Every component instance label, the idle label and the interrupt label
      * have every authority over themselves
      *)
     {edge. edge_subject edge ∈ {idle_label, irq_label} ∪
              fst ` set (components (composition spec)) ∧
            edge_subject edge = edge_object edge} ∪

     (* Senders on seL4RPCSimple connections *)
     {edge. ∃(name, conn) ∈ set (connections (composition spec)).
              conn_type conn = seL4RPCSimple ∧
              edge_object edge = name ∧
              edge_subject edge = fst (conn_from conn) ∧
              edge_auth edge ∈ {Reset, SyncSend}} ∪

     (* Receivers on seL4RPCSimple connections *)
     {edge. ∃(name, conn) ∈ set (connections (composition spec)).
              conn_type conn = seL4RPCSimple ∧
              edge_object edge = name ∧
              edge_subject edge = fst (conn_to conn) ∧
              edge_auth edge ∈ {Receive, Reset, SyncSend}} ∪

     (* Senders on seL4AsynchNative connections *)
     {edge. ∃(name, conn) ∈ set (connections (composition spec)).
              conn_type conn = seL4AsynchNative ∧
              edge_object edge = name ∧
              edge_subject edge = fst (conn_from conn) ∧
              edge_auth edge ∈ {AsyncSend, Reset}} ∪

     (* Receivers on seL4AsynchNative connections *)
     {edge. ∃(name, conn) ∈ set (connections (composition spec)).
              conn_type conn = seL4AsynchNative ∧
              edge_object edge = name ∧
              edge_subject edge = fst (conn_to conn) ∧
              edge_auth edge ∈ {Receive, Reset}}"
```

It is observable from this definition how the authority between labels is derived from the same intuition the system designer has about the relationship between their component instances. The result is a generated authority graph that feels very familiar to the system designer, which is an invaluable trait when it comes to reasoning about their system. To demonstrate this, the authority graph of the system from Figure 7.1 is visualised in Figure 7.3.

A side effect of producing a set rather than a single policy is that the output of the generator function represents the most fine-grained desirable policy. It is possible that the system designer does not need this level of detail. For example, the system may contain two component instances whose isolation from each other is irrelevant. In such cases, it is straightforward for the designer to define a more coarse policy by merging certain labels and providing a mapping from one label set to the other. This new policy can be trivially proved valid by showing its authority graph is a superset of the original's authority graph.

We have explained how the policy for a CAmkES system is generated, but not why the above generation logic is correct. For this, we have manually proven a further lemma that, if the generator function produces a CapDL specification and a policy, the specification always refines the policy.

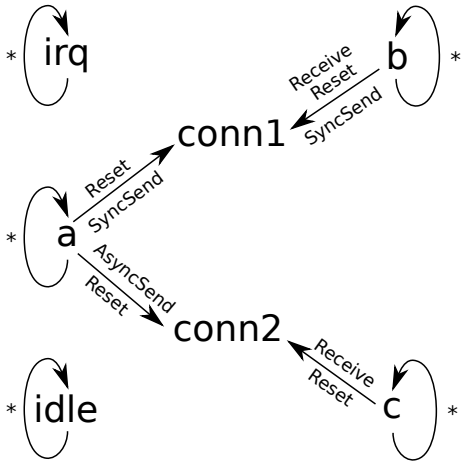

Figure 7.3: Authority graph for an example system involving three components

```
lemma generated_pcs_refined:
  "⟦extended_generate spec extra irqs = Some (capdl, pases); pas ∈ pases⟧
    ⟹ pcs_refined pas capdl"
```

In this lemma statement, `pcs_refined` is a relation defined in other work (Klein et al. 2014) describing refinement between an arbitrary CapDL specification and a policy. The effect of this lemma is to guarantee consistency in the outputs of the generator function. Moreover, it obviates the need for the system designer to construct their own policy or verify its correctness.

To emphasise this, the foundation we have defined allows a system designer to input only their component architecture, and have a guaranteed-accurate information flow policy generated for them. This significantly reduces impediments to reasoning about information flow in a component-based system, and even opens the possibility for proving simple properties automatically.

7.8 Example

To demonstrate the utility of this formalism, we recapitulate our familiar example of a system with two connected component instances.

```
procedure Simple {
  int echo_int(in int i);
  int echo_parameter(in int pin, out int pout);
}

component Client {
  control;
  uses Simple s;
}

component Echo {
  provides Simple s;
}

assembly {
  composition {
```

```
17      component Echo echo;
18      component Client client;
19
20      connection seL4RPCSimple simple(from client.s, to echo.s);
21    }
22  }
```

Recall that this system is depicted in Figure 5.4.

Translation into the static architecture model is automatic, as is generation of the corresponding CapDL specification and proof of its correspondence to the output of the Isabelle/HOL function we described previously. The resulting policy delivered to the designer contains the following authority graph.

```
1  definition
2    authgraph :: "(label × auth × label) set"
3  where
4    "authgraph ≡
5      {edge. edge_subject edge = edge_object edge ∧
6             edge_subject edge ∈ {''client'', ''echo'', idle_label, irq_label}} ∪
7      {(''client'', Reset, ''simple''),
8       (''client'', SyncSend, ''simple''),
9       (''echo'', Receive, ''simple''),
10      (''echo'', Reset, ''simple''),
11      (''echo'', SyncSend, ''simple'')}"
```

This generated graph shows a set of edges that should match the designer's intuition for the system they have built. In this case, the graph contains a self-reflexive edge from every component instance label to itself, as well as the same for the idle thread and interrupt labels (lines 5-6). The remaining set shows the authorities for resetting the state of an object (`Reset`) and synchronous sending (`SyncSend`) from `client` to the connection label `simple` (lines 7-8) and authorities for the same and additionally receiving (`Receive`) from `echo` to `simple` (lines 9-11). This authority graph is depicted in Figure 7.4.

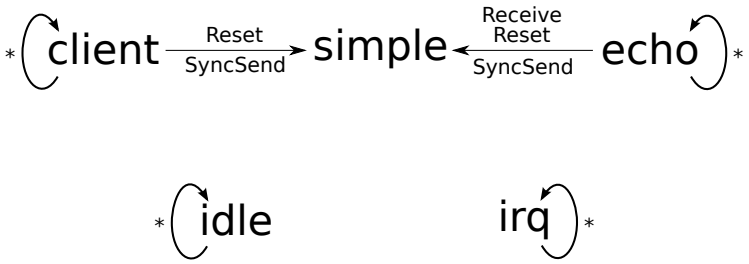

Figure 7.4: Authority graph of a simple component system

From here, it is possible to prove properties of the architecture. For example, it can be proven that `simple` has no authority over `client`.

```
1  lemma no_back_channel: "(''simple'', auth, ''client'') ∉ authgraph"
2    by (simp add:authgraph_def)
```

Note that the proof of this lemma is trivial and many other safety and security properties will be similarly simple. This is due to the simple set type of the authority graph. For a constrained class of such lemmas, it is possible to automate such proofs. We have not attempted to do so, as it would require classifying and

constraining the class of lemmas users would be expected to want, but it would be a natural extension to this line of work.

This section has summarised the expected work flow for proving information flow properties about a CAmkES system. The formalism of this chapter can similarly be used as a basis for proving integrity properties and for guaranteeing memory separation, the latter being necessary for future work to further cement the connection between the model of Chapter 6 and seL4.

The brevity of the steps involved in going from a system architecture to the proven property of this section reflect the extensive automation. The details and internal structure of generated proofs are not a concern of the designer. In the next section, we elaborate on the motivation for trusting these unseen artefacts.

7.9 Trust

The process we have demonstrated allows the designer to prove information flow properties about their component systems. Having proven such properties, it is desirable to have a guarantee that the model on which they were proven matches the running system.

With the exception of the initial translation from textual architecture description to an Isabelle/HOL term built on the formalism from Chapter 4, all steps are taken within the theorem prover. In particular, the proof of correspondence between the output of the Isabelle/HOL generator function and the concrete CapDL specification of a given system is replayed within the theorem prover. Thus we do not need to trust the logic that generates this proof, as it will simply fail when replayed if incorrect. The only element included in our trusted computing base here is the theorem prover itself.

The initial architecture translation that occurs outside the theorem prover can be trusted to be correct or its output can be inspected. The correspondence between input and output formats is close enough that a manual validation is possible. An alternate approach would be to perform this translation in the opposite direction. Defining an Isabelle/HOL function that accepted a term describing the architecture and produced a textual representation suitable for consumption by CAmkES' code generator would permit this. The Isabelle/HOL function could then be proven to preserve some given semantics during this translation. We have not taken this approach, but we present it here as an alternative means of gaining confidence in the correctness of the translation. In practice, infidelities in the existing translation mechanism were always immediately apparent to users when their expected properties could not be proven.

An alternate perspective with regard to trusting the architecture translation is that the ADL description of a component system is no longer a critical component of system correctness. Returning to Figure 2.10 and depicting the generated artefacts discussed in this chapter separately, we have Figure 7.5. By viewing the generated architecture definition as the canonical representation of the system, the lack of a formal connection between the original ADL description and this generated definition becomes irrelevant. As discussed previously, the correspondence between this generated architecture definition and the initialisation specification is proven entirely within the theorem prover. This removes the need to trust the translation process altogether.

7.10 Scalability

The example we demonstrated in Section 7.8 involved only two component instances, and it is natural to ask how this process scales to larger component systems. The generated proof for the example given takes around 20 minutes to replay on an Intel Core i7-4770 with 32GB of RAM, running Linux 3.13.0 and Isabelle 2015. The

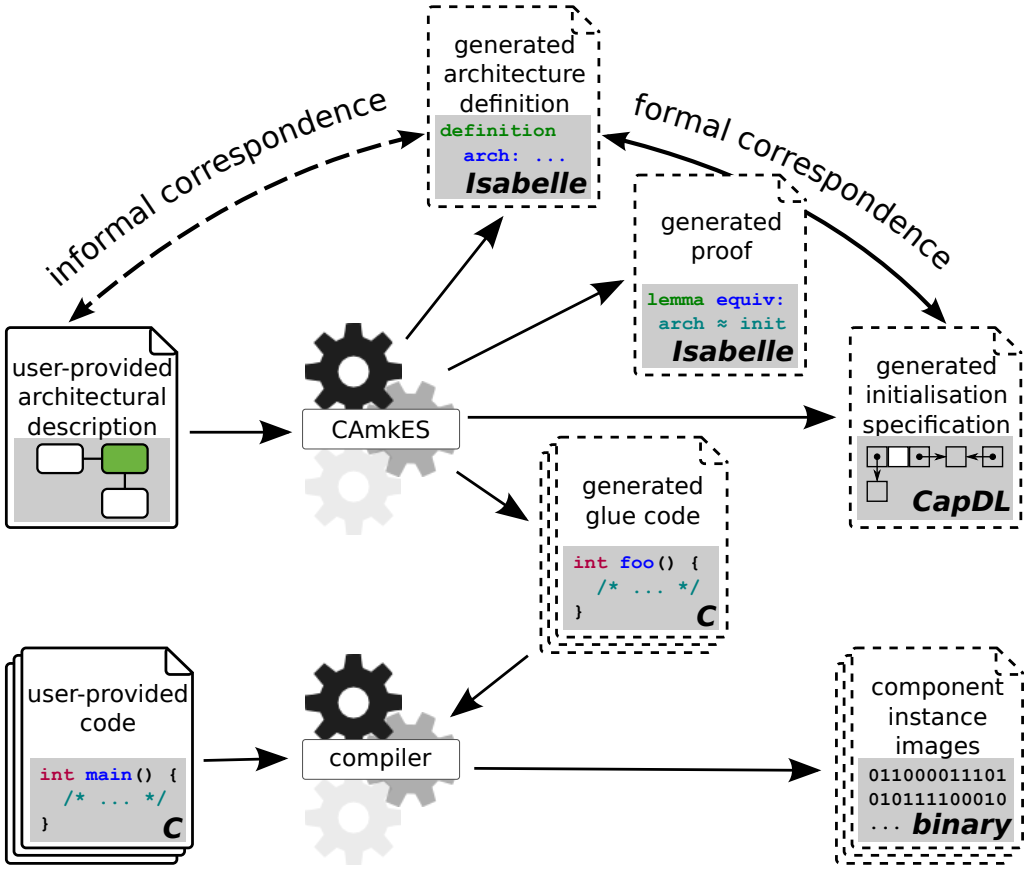

Figure 7.5: Informal and formal correspondence between architecture artefacts

proof has not been heavily optimised and could easily be sped up by a factor of two. Its current runtime is dominated by proofs about capabilities to address space objects; specifically, that there is a single capability to each frame and page table.

A weakness of our current implementation is that the address space objects are represented in a collection as a single Isabelle/HOL term, leading this proof to be quadratic in the number of component instances in the system. However, this is not a fundamental bound. By requiring a single capability to each frame and page table, we implicitly constrain the address space objects to be partitioned into disjoint directed acyclic capability graphs, one for each component instance. Representing these disjoint graphs as separate Isabelle/HOL terms would diminish the proof goal to demonstrating that they are indeed disjoint and that each form a tree, rooted at a page directory.

By grouping address space objects by component instance at the time of their creation, it is possible to assign them numerically contiguous identifiers. It is then possible to prove collections of objects disjoint by way of numeric ranges. This design admits an alternate proof strategy that scales linearly in the number of component instances. While this is not the approach in our current implementation, we note this possibility here as a justification that a theoretical upper bound on the scalability of our technique is linear in the number of component instances.

7.11 Summary

This chapter has demonstrated a technique for automatically deriving an access control policy for a component-based system. The derived policy is automatically proven to correspond to a fine-grained specification of kernel objects and their capabilities. By treating information flow at the granularity of component instances and connections, the generated policy matches the system designer's intuitions and forms a natural environment for proving safety and security properties.

The automation we provide allows a system designer to seamlessly transition from describing a component system architecture to proving properties about that architecture. In bridging these environments, we enable iteration on an initial design while prototyping desired safety and security properties and observing how architecture changes affect the formal properties of a system. In doing so, system designers gain a better understanding of the implications of their architectural patterns.

Chapter 8

Example System Verification

Thus far we have introduced four separate formal models for reasoning about component based systems: a static architecture model (Chapter 4), a dynamic behavioural model (Chapter 5), code-level correctness proofs (Chapter 6) and access control policy generation (Chapter 7). In this chapter, we demonstrate how to apply these four in concert to a realistic prototype system and highlight the benefits each deliver for the system designer. In doing so, we aim to clarify the utility and application of our work and indicate its effectiveness for generalised component-based systems.

8.1 Multi-level Terminal

To demonstrate our techniques, we have built a prototype of a multi-level terminal. A multi-level terminal is a single software system that is designed to be used for handling information with differing security requirements. Data at different levels of secrecy or security are handled by separate, isolated domains. Information flow between the domains is restricted and driven by the system's security policy. The concept of a multi-level terminal stems from ideas that originated in the design of separation kernels (Alves-Foss et al. 2006; Rushby 1981; Rushby 1984).

The prototype we have implemented contains four component instances: a trusted source of input, a trusted display and two instances handling separate security levels. We designate the security levels "low" and "high," with information flow permitted from low to high, but not vice versa. It is common for multi-level terminals to have more than two domains and even a lattice-based security policy (Denning 1976). However, a pair of domains are sufficient for demonstrating the application of our formalisms, and it should be clear how to generalise the forthcoming steps to a greater number of domains. The system architecture is depicted in Figure 8.1.

The `input` component instance feeds character data to either `low` or `high`. When it receives a designated keyboard input, the tab character, it switches from sending keystrokes from one of these to the other. The `low` and `high` instances process their received keystrokes by either outputting data to the `gui` component instance or making changes to their local state. The local state of `high` is considered secret, in that `low` should not be able to infer it. The `gui` component instance receives screen painting directives from both `low` and `high` and represents these on dedicated regions of a display, one for each source domain. Synchronous endpoints imply bidirectional information flow (Murray et al. 2012), so we need to trust `input` and `gui`. If we do not trust these component instances, they could maliciously leak information from one domain to another. The design we have described thus far prevents information flow between `low` and `high`. However,

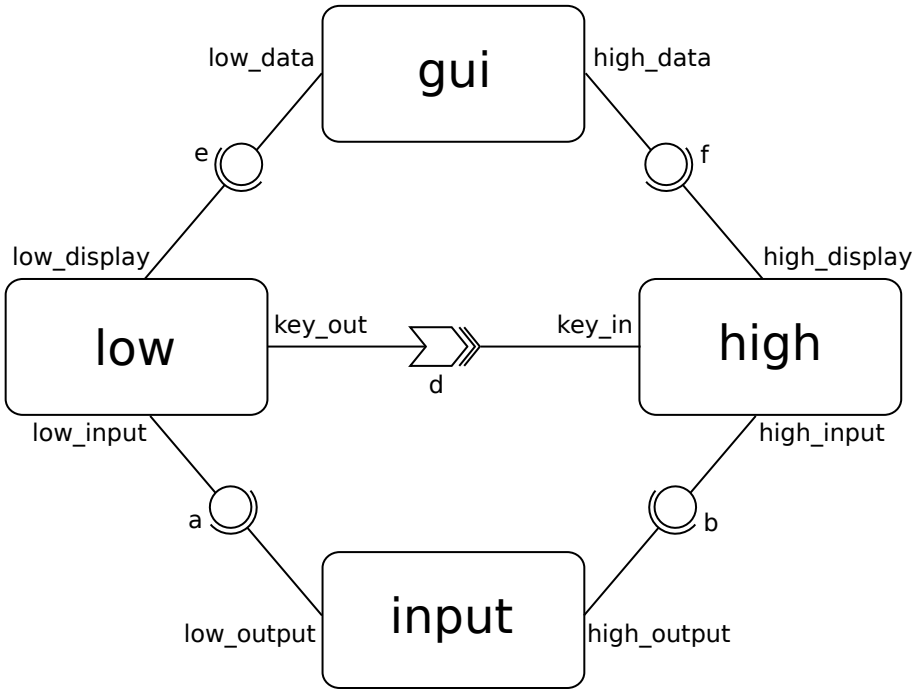

Figure 8.1: Multi-level terminal prototype

we additionally allow `high` to observe when `low` is receiving data via an event emitted by `low`. The final architecture permits unidirectional information flow; from `low` to `high`, but not in the opposite direction. An unlabelled, coarse graph of the intended information flow within this system is depicted in Figure 8.2.

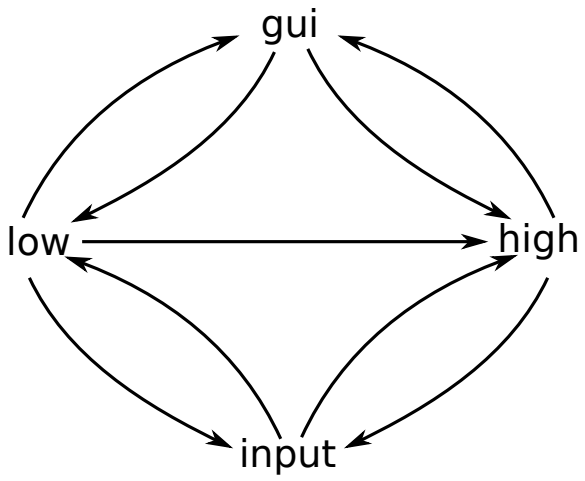

Figure 8.2: Information flow in the multi-level terminal prototype

The full specification of this system is as follows.

```
procedure Paint {
  /* Write a character at the current cursor position in caller's region */
  int put_char(in char c);

  /* Delete all characters from the caller's region */
  void clear(void);

  /* Move the caller's cursor to the given position */
  int move(in unsigned int x, in unsigned int y);
}

procedure Data {
  /* Send a key stoke */
  void send_key(in char c);
}

component Input {
  control;
  uses Data low_output;
  uses Data high_output;
}

component Low {
  provides Data low_input;
  uses Paint low_display;
  emits Keypress key_out;
}

component High {
  control;
  consumes Keypress key_in;
  provides Data high_input;
  uses Paint high_display;
}

component GUI {
  control;
  provides Paint low_data;
  provides Paint high_data;
}

assembly {
  composition {
    component Input input;
    component Low low;
    component High high;
    component GUI gui;

    connection seL4RPCSimple a(from input.low_output, to low.low_input);
    connection seL4RPCSimple b(from input.high_output, to high.high_input);
    connection seL4AsynchNative d(from low.key_out, to high.key_in);
    connection seL4RPCSimple e(from low.low_display, to gui.low_data);
    connection seL4RPCSimple f(from high.high_display, to gui.high_data);
  }
}
```

When running, the multi-level terminal appears as in Figure 8.3. The two display domains for low and high information sit side-by-side in separate windows. A cursor indicates which window is currently receiving character input. The status bar at the bottom of each window shows the currently active input mode, in the style of the Vim editor, in that window. Additionally the status bar of the high window has an upper case "X" that blinks to indicate when the low window is receiving character input.

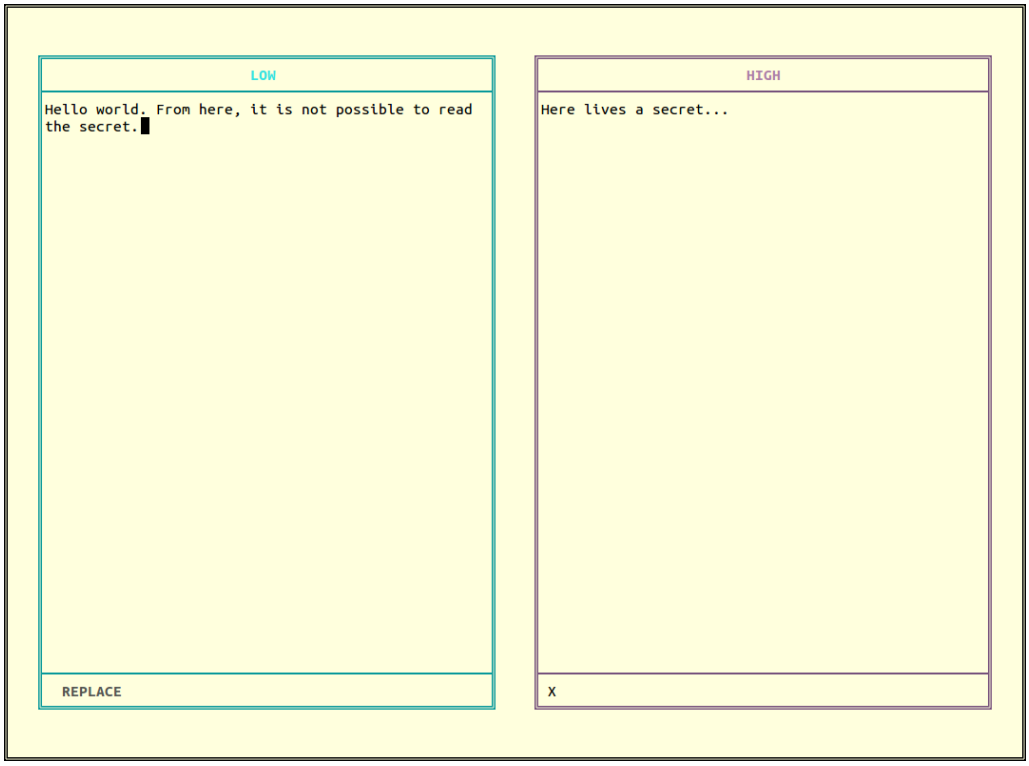

Figure 8.3: Multi-level terminal running

This system has some intuitive desirable correctness properties. For instance, information should not be able to flow from `high` to `low`. That is, `low` should not be able to learn anything about `high`. To believe what is shown to us, we would also like to know that `low` and `high` are only able to affect their respective regions of the display, not each other's. With these intentions in mind, we now consider how the formalisms of our work aid us in providing formal guarantees that we have achieved our goals.

8.2 Architectural Correctness

The specification from Section 8.1 is translated automatically by CAmkES into an Isabelle/HOL theory built on the model introduced in Chapter 4. The definition of the composition matches its textual equivalent in the original specification.

```
definition
  composition' :: composition
where
  "composition' ≡ ⦇
    components = [(''input'', input), (''low'', low), (''high'', high),
      (''gui'', gui)],
    connections = [(''a'', a), (''b'', b), (''d'', d), (''e'', e), (''f'', f)] ⦈"
```

The rest of the translated specification has been elided, but is reproduced in full in Appendix C. Proving that this system is wellformed is, as expected, fully automatic.

```
lemma wf_assembly': "wellformed_assembly assembly'"
  by code_simp
```

This lemma validates the consistency of the specification we have provided, and we now know that this system can be instantiated correctly by CAmkES.[1] From this representation, we could prove some architectural properties. However, in such a tightly connected system, the properties we can prove are limited without assuming information flow properties of the connections. Instead, we look to our other formalisms for showing such properties in a more rigorous manner.

8.3 Dynamic Behaviour

To obtain a representation of the system's behaviour at runtime, we can turn to the formalism of Chapter 5. Again, CAmkES generates the Isabelle/HOL definitions for this system automatically. The connections and component instances are enumerated in two data types.

```
datatype channel
  = a
  | b
  | d
  | e
  | f

datatype inst
  = input
  | low
  | high
  | gui
  | "d_e"
```

Note that we have an artificial component instance, d_e, representing the event connection in the system. Component interface elements are emitted as generated definitions. For example, the `put_char` method of the `low_display` interface of `Low` appears as follows.

```
definition
  Call_Low_low_display_put_char :: "(Low_channel ⇒ channel) ⇒
    ('cs local_state ⇒ char) ⇒ ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Call_Low_low_display_put_char ch10' c_p embed_ret ≡
     Request (λs11'. {⦇ q_channel = ch10' Low_low_display,
                        q_data = Call 0 [Char (c_p s11')] ⦈}) discard ;;
     Response (λs12' s13'. case q_data s12' of
         Return s14' ⇒ {(embed_ret s13' (case hd s14' of Integer s15' ⇒ s15'),
                         ⦇ a_channel = ch10' Low_low_display,
                           a_data = Void ⦈)}
       | _ ⇒ {})"
```

[1] It is tempting to conclude that, as the system boots and runs as expected, we already had this guarantee. However, recall from Chapter 4 that we permit CAmkES to generate faulty code in the case of an input that is not wellformed. By proving the system's specification wellformed, we guarantee that the system we witness running has well-founded semantics and has been instantiated in conformance with its specification.

This definition takes a parameter `ch10'`, indicating the channel (connection) to send on, c_P, a projection function to retrieve the character to send from the local state, and `embed_ret`, an embedding function to insert the return value back into the local state. The character to send is extracted and transmitted in the `Request` invocation (lines 7-8). The return value is received and inserted in the `Response` invocation (lines 9-13).

The initial state of the system – also generated by CAmkES – contains each of the component instances and their untrusted definitions. Recall that these can be overridden by trusted definitions post hoc if desired.

```
1  (* Global initial state *)
2  definition
3    "gs₀" :: "(inst, channel, 'cs) global_state"
4  where
5    "gs₀ p124' ≡ case trusted p124' of Some s125' ⇒
6        Some s125'
7      | _ ⇒ (case p124' of
8            input ⇒ Some (input_untrusted, Component init_component_state)
9          | low ⇒ Some (low_untrusted, Component init_component_state)
10         | high ⇒ Some (high_untrusted, Component init_component_state)
11         | gui ⇒ Some (gui_untrusted, Component init_component_state)
12         | dₑ ⇒ Some (dₑ_instance, init_event_state))"
```

We elide the remainder of this generated theory, but it is included in full in Appendix D. As with the untrusted example we demonstrated in Chapter 5, there are only limited properties we can show in the absence of more precise component instance definitions. Reusing the predicates introduced in Chapter 5, it is possible, for example, to prove that `input` and `gui` cannot directly communicate.

```
1  lemma
2    "¬ (∃c. communicates_on c input_untrusted ∧ communicates_on c gui_untrusted)"
3    by (auto simp:communicates_on_def input_untrusted_def
4                    Input_untrusted_def gui_untrusted_def
5                    GUI_untrusted_def UserStep_def ArbitraryRequest_def
6                    ArbitraryResponse_def)
```

We cannot prove that `high` cannot directly communicate information to `low` on the generalised untrusted definitions. To achieve this, we need to give trusted definitions of the component types, `High` and `Low`, that indicate a more precise semantics of the seL4AsynchNative connector. We replicate their untrusted definitions, but replace the `ArbitraryRequest` and `ArbitraryResponse` corresponding to connection `d` with the possible event actions.

```
1  definition
2    High_trusted :: "(High_channel ⇒ channel) ⇒
3      (channel answer, channel question, unit local_state) com"
4  where
5    "High_trusted ch ≡ LOOP (
6         UserStep
7       ⨿ ArbitraryRequest (ch High_high_display)
8       ⨿ ArbitraryResponse (ch High_high_display)
9       ⨿ ArbitraryRequest (ch High_high_input)
10      ⨿ ArbitraryResponse (ch High_high_input)
11      ⨿ EventPoll (ch High_key_in) (λs _. s)
12      ⨿ EventWait (ch High_key_in) (λs _. s))"
13
```

```
definition
  Low_trusted :: "(Low_channel ⇒ channel) ⇒
    (channel answer, channel question, unit local_state) com"
where
  "Low_trusted ch ≡ LOOP (
       UserStep
     ⌊⌋ ArbitraryRequest (ch Low_low_display)
     ⌊⌋ ArbitraryResponse (ch Low_low_display)
     ⌊⌋ ArbitraryRequest (ch Low_low_input)
     ⌊⌋ ArbitraryResponse (ch Low_low_input)
     ⌊⌋ EventEmit (ch Low_key_out))"
```

Note that our trusted behaviour definitions may still over-approximate both components' behaviour with respect to this connection. We only narrow the components' behaviour to the actions seL4AsynchNative allows, not further to only the actions they take. In general, this limited narrowing – only constraining behaviour minimally to prove the required property – is good practice. By being conservative in our constraints we reduce the risk of over-constraining components to the point at which their definition is no longer a valid abstraction. That is, we minimise the likelihood of removing an abstract behaviour that the implementation *does* exhibit.

We can now recapitulate the untrusted definitions of the instances, `high` and `low`, referencing these new trusted component definitions.

```
definition
  high_trusted :: "(channel answer, channel question, unit local_state) com"
where
  "high_trusted ≡ High_trusted (λs. case s of
     High_high_display ⇒ f | High_high_input ⇒ b | High_key_in ⇒ d)"

definition
  low_trusted :: "(channel answer, channel question, unit local_state) com"
where
  "low_trusted ≡ Low_trusted (λs. case s of
     Low_low_display ⇒ e | Low_low_input ⇒ a | Low_key_out ⇒ d)"
```

Overriding the untrusted definitions with these alternatives makes it possible to prove that `high` never directly sends to `low`.

```
lemma "¬ (∃c. receives_on c low_trusted ∧ sends_on c high_trusted)"
  by (auto simp:low_trusted_def Low_trusted_def high_trusted_def
                High_trusted_def UserStep_def ArbitraryRequest_def
                ArbitraryResponse_def
                EventPoll_def EventWait_def EventEmit_def)
```

Note that, in accomplishing this, we have assigned semantics to the connection between `low` and `high` that are intended, but for which we do not have a formal justification. That is, the proof assumes the correctness of generated code and initialisation specification. These two assumptions are addressed by our formalisms from Chapter 6 and Chapter 7, which we now go on to leverage.

8.4 Code Generation

Amongst the properties of interest in this system, is that `low` can only ever write to its dedicated region of the display. Whilst this is intended to be enforced by the `gui` component instance, cursor positioning and

character writes are directed via the incoming `low_data` interface of `gui`. This correctness property relies on the behaviour of both the glue code and component code. We can show this in a code-level proof using the formalism from Chapter 6.

The `gui` component instance writes character data to the screen using an external function.

```
/* Display the character, c, on the screen at location (x, y) */
extern void putch(char c, const char *style, unsigned int x, unsigned int y);
```

To write data to the low region of the display, this is invoked by the implementation of the incoming interface method, `low_data_put_char`.

```
int low_data_put_char(char c) {
  putch(c, low_y >= LOW_Y + LOW_HEIGHT - 2 ? A_FG_K A_BOLD : A_RESET,
    low_x, low_y);
  return 0;
}
```

It can be observed that this function always writes character data at the location given by a pair of global variables, `low_x` and `low_y`. These globals and another set that track the cursor position outside the status bars are used to store the state of the `gui` component instance.

```
/* The last position of the cursor in either domain excluding status bars */
static unsigned int last_x;
static unsigned int last_y;

/* The current position of the low cursor */
static unsigned int low_x = LOW_X + 1;
static unsigned int low_y = LOW_Y + 3;
```

The position of the low cursor, given by `low_x` and `low_y` is updated via the `low_data_move` interface method.

```
int low_data_move(unsigned int x, unsigned int y) {

  /* Bounds check the coordinates */
  if (x <= LOW_X || x >= LOW_X + LOW_WIDTH - 1)
    return -1;
  if (y <= LOW_Y + 2 || y >= LOW_Y + LOW_HEIGHT - 1)
    return -1;

  low_x = x;
  low_y = y;

  /* If the cursor was moved somewhere outside the status bar,
   * update the last position data
   */
  if (!(y >= LOW_Y + LOW_HEIGHT - 3)) {
    last_x = x;
    last_y = y;
  }

  return 0;
}
```

Recall this function is invoked by glue code in the `gui` component instance. This glue code is, in turn, invoked by glue code in the `low` component instance that is itself called by hand written code in `low`.

Thus, to prove our desired property, we must show that an invocation of `low_display_move` in `low` can only ever result in the low cursor being positioned within the low region of the display.[2] Recall from Chapter 6 that achieving a proof over such heterogeneous code requires manually proving a desired precondition and postcondition of the callee's implementation. In this case, we define them identically, as a predicate constraining valid low cursor coordinates.

```
definition
  valid_coordinates :: "lifted_globals ⇒ bool"
where
  "valid_coordinates s ≡
    low_x s ≥ scast LOW_X ∧ low_x s ≤ scast (LOW_X + LOW_WIDTH - 1) ∧
    low_y s ≥ scast LOW_Y ∧ low_y s ≤ scast (LOW_Y + LOW_HEIGHT - 1)"
```

The extraneous `scast` applications in this definition are simply to convert signed integer values to unsigned integer values. We take an inductive approach in this verification, proving a base case, followed by an inductive step on system states. To show that the system begins in a valid state, we are able to prove the initial, statically assigned values of `low_x` and `low_y` satisfy this predicate.

```
lemma valid0:
  "⟦low_x s = scast (LOW_X + 1); low_y s = scast (LOW_Y + 3)⟧
     ⟹ valid_coordinates s"
```

The inductive step case to be proven is local correctness of `low_data_move`. Namely, if the predicate holds over the current state, it then holds over the resulting state from running `low_data_move`. Using the predicate, `valid_coordinates`, we have defined, this can be accomplished using standard techniques.

```
lemma valid: "{|valid_coordinates|} low_data_move x y {|λ_. valid_coordinates|}!"
```

Note that, in this lemma, we do not claim the function returns `0` (the return value is ignored in the postcondition). We want to ensure all paths of execution, success or otherwise, result in valid low coordinates.

The correctness lemma we need to lift an arbitrary precondition and postcondition over a remote invocation of the method `move` for the connection `e` is generated automatically.

```
lemma low_display_low_data_move_equiv_wp:
  "∀s0. {|λs. s = s0 ∧ inv s ∧ move_P s p0 p1|}
          do low_display_move_marshal p0 p1;
             low_data_move_internal;
             low_display_move_unmarshal
          od
        {|λr s. inv s ∧ move_Q s0 s r p0 p1|}!"
```

Recall from Chapter 6 that, in order to use this lemma, we must interpret the locale within which it is contained. To do so we need to provide seven parameters: a symbol table mapping strings to addresses and a precondition and postcondition for each method of connection `e`. The symbol table is required only for proving distinctness

[2] A more thorough proof of this property would also involve demonstrating that there are no additional ways to affect `low_x` and `low_y`, either through other methods of `low`'s `low_display` interface, or through other connections. Such a proof is possible within our framework, but the proof consists of an extensive enumeration of trivial cases and is uninteresting. For this reason, we exclude it from our example.

of global variables, a property that is guaranteed by the linker and program loader, and is not something our work is attempting to address. So, for now, we define an artificial symbol table that gives each relevant global a distinct address. In future the correctness of the linker and loader could be addressed by techniques from other ongoing work (Sewell, Myreen, and Klein 2013).

```
definition
  symtab :: "string ⇒ 32 word"
where
  "symtab s ≡ if s = ''put_char_c_1'' then 0x1000
         else if s = ''put_char_c_2'' then 0x1001
         else if s = ''put_char_c_3'' then 0x1002
         else if s = ''move_x_1'' then 0x1004
         else if s = ''move_x_2'' then 0x1008
         else if s = ''move_x_3'' then 0x100c
         else if s = ''move_y_1'' then 0x1010
         else if s = ''move_y_2'' then 0x1014
         else if s = ''move_y_3'' then 0x1018
         else 0"
```

We can now go on to interpret the locale by providing the preconditions and postconditions. We use trivial predicates for the methods that are not relevant to our current aim.

```
interpretation e_pruned_glue symtab
  "λ_ _. True" "λ_ _ _ _. True" "λ_. True" "λ_ _. True"
  "λs _ _. valid_coordinates s" "λs s' _ _ _. valid_coordinates s'"
```

The proof of the locale interpretation is straightforward and yields a final lemma that guarantees our supplied precondition and postcondition hold over a remote invocation of the `move` method.

```
lemma correct:
   "∀s0. {|λs. s = s0 ∧ inv s ∧ valid_coordinates s|}
       do low_display_move_marshal x y;
          low_data_move_internal;
          low_display_move_unmarshal
       od
       {|λr s. inv s ∧ valid_coordinates s|}!"
```

In this section we have selectively chosen a property of one method to verify, but it is possible to repeat the steps we have taken to prove properties of other methods or, more sensibly, to prove multiple properties within the same interpretation step. This approach scales linearly in the number of methods within an interface and across the number of connections in the system.

8.5 Policy Generation

What remains to demonstrate on this system is that the initial configuration of the system is correct and that it has the security properties intended. In the case of this system, the property we would like to show is that `high` does not send information to `low`. This can be proven within the formalism we introduced in Chapter 7.

As for the system shown in Chapter 7, a proof that the output of our Isabelle/HOL function corresponds to the CapDL specification produced for this system is generated automatically.

```
lemma final:"∃pases. extended_generate spec extra irqs = Some (final, pases)"
```

The authority graph contained within the policy produced by this function appears as follows. Recall that we previously proved that any policy produced by this function is refined by the CapDL specification produced by the function.

```
definition
  authgraph :: "(label × auth × label) set"
where
  "authgraph ≡
    {edge. edge_subject edge = edge_object edge ∧
           edge_subject edge ∈ {''gui'', ''high'', ''input'', ''low'',
                               irq_label, idle_label}} ∪
    {(''input'', Reset, ''a''), (''input'', SyncSend, ''a''),
     (''input'', Reset, ''b''), (''input'', SyncSend, ''b''),
     (''low'', Receive, ''a''), (''low'', Reset, ''a''),
     (''low'', SyncSend, ''a''), (''low'', Reset, ''e''),
     (''low'', SyncSend, ''e''), (''low'', Reset, ''d''),
     (''low'', AsyncSend, ''d''), (''high'', Receive, ''b''),
     (''high'', Reset, ''b''), (''high'', SyncSend, ''b''),
     (''high'', Reset, ''f''), (''high'', SyncSend, ''f''),
     (''high'', Receive, ''d''), (''high'', Reset, ''d''),
     (''gui'', Receive, ''e''), (''gui'', Reset, ''e''),
     (''gui'', SyncSend, ''e''), (''gui'', Receive, ''f''),
     (''gui'', Reset, ''f''), (''gui'', SyncSend, ''f'')}"
```

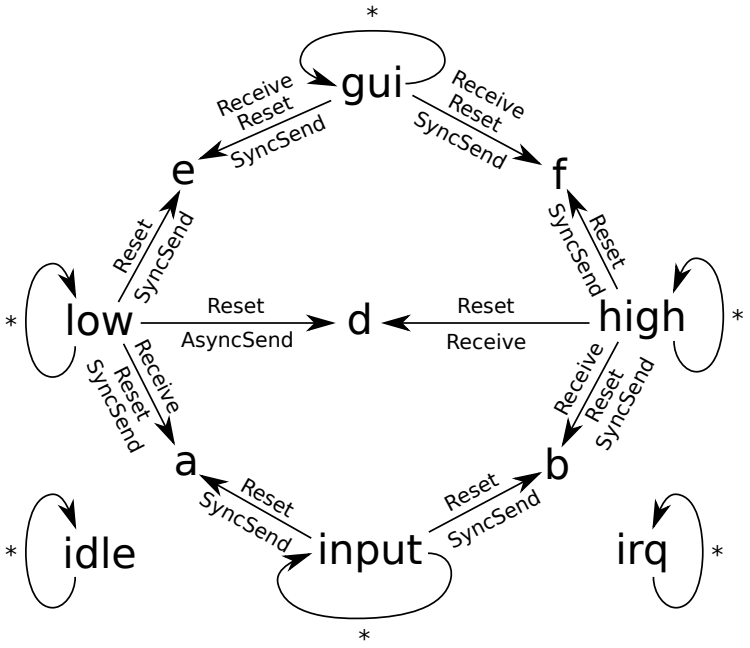

Figure 8.4: Fine-grained authority graph of multi-level terminal prototype

This graph is depicted in Figure 8.4. Though it is significantly more dense than the authority graphs we have seen thus far, it is still composed of simple data types and thus reasoning about this term is still straightforward. For example, we can prove that `high` does not directly transmit information to `low`.

```
lemma no_send_high_low: "¬ (∃edge ∈ authgraph.
  edge_subject edge = ''high'' ∧ edge_object edge = ''low'')"
  by (clarsimp simp:authgraph_def)
```

We could prove a stronger, transitive information flow property about `high` and `low`, but it would require assuming behaviour of `gui` and `input`. This is because synchronous endpoints permit bidirectional information flow, allowing `high` to send information to `low` via a cooperating `gui` or `input` implementation. A transitive information flow property without assumptions would require a modified architecture that avoided the use of synchronous endpoints at trust boundaries. Instead, we have used a more permissive system architecture in this example to ease explanation and understanding.

While the proof of this property was uncomplicated, in general the complexity of proofs within this formalism will depend heavily on the complexity of the component system architecture. However, the correspondence between the generated authority graph and the system designer's intuition ensures this is controllable. That is, thoughtful design of a system architecture can help manage verification complexity. This is a pleasant side effect of the precision with which we model information flow within a CAmkES system.

8.6 Summary

Prior to this chapter, we have independently introduced four formalisms for reasoning about component-based systems. In this chapter, we have shown how to leverage the combination of these four in pursuit of component system correctness. We have emphasised how closely the generated formal artefacts correspond to the system designer's intuition and stressed the ease with which eventual manual proofs can be accomplished. The construction of the example system and its verification took approximately half a day, indicative of the general effort-to-outcome ratio. The assurance guarantees we have achieved for this system, though not deep, are relatively powerful for the effort invested. With further effort it is possible to achieve comprehensive, code-level verification of a component-based system. Overall this discussion highlights the functionality and practical utility of the verification techniques that are the subject of this work.

Chapter 9

Limitations and Future Work

This thesis discusses work aimed at guaranteeing a particular form of correctness in component-based systems. While individual chapters have noted specific technical limitations, this chapter outlines the high level boundaries of our work. In making these limitations explicit, we also provide a discussion of how these could be addressed in future.

9.1 Connection to seL4

The functional correctness properties we demonstrate of the CAmkES component platform are predicated on the correctness of seL4. In particular, we currently axiomatise the semantics of seL4 system calls that are invoked through a low level wrapper to the kernel's API known as "libsel4." While the abstract specification of seL4 precisely describes the behaviour of each system call, these definitions lie within a dense, generalised definition of the kernel's behaviour during an arbitrary invocation. Knowing the precise state of the system and exact parameters to any system call we invoke, it is possible to form specialised definitions of kernel behaviour for each invocation. These can be much more concise and manageable by, for example, eliding error handling logic guarded by a condition that we know will be false.

Extracting these latent, specialised system call semantics is not straightforward.[1] In addition, the control flow of a system call cannot precisely be modelled as a function call. After completing the action requested by a system call, the context switch back to userspace is preceded by seL4's scheduler choosing a thread to run. Thus in-between a system call invocation and its successful return, other userspace threads may execute, changing the global state of the system. CAmkES systems are sufficiently constrained that we know a priori that the actions of these intermediate threads cannot affect the caller, but a semantics must still capture these actions to be precise. The cooperative requirements of threads on each other would fit elegantly with a Rely/Guarantee style of reasoning (Jones 1983; Owicki and Gries 1976). However, rather than attempting to form a further complex, constrained semantics, we observe that this class of concurrency is a subset of a more general problem that we believe is better dealt with by other orthogonal ongoing work (Andronick, Lewis, and Morgan 2015).

For now, we axiomatise the system call stubs from libsel4, that present a natural cut point. These functions are responsible for mediating between the C calling convention and seL4's ABI. They are called from CAmkES glue code and are implemented in libsel4 by platform-specific assembly instructions. As an example, the definition of `seL4_Call` on the ARM platform is as follows.

[1] To gain a more accurate picture of the complexity involved and what such a specification would look like, the reader is referred to Section 7.4 of prior work on initialisation verification (Boyton 2014).

```
seL4_MessageInfo_t seL4_Call(seL4_CPtr dest, seL4_MessageInfo_t msgInfo) {
  register seL4_Word destptr asm("r0") = (seL4_Word)dest;
  register seL4_MessageInfo_t info asm("r1") = msgInfo;

  /* Load beginning of the message into registers */
  register seL4_Word msg0 asm("r2") = seL4_GetMR(0);
  register seL4_Word msg1 asm("r3") = seL4_GetMR(1);
  register seL4_Word msg2 asm("r4") = seL4_GetMR(2);
  register seL4_Word msg3 asm("r5") = seL4_GetMR(3);

  /* Perform the system call */
  register seL4_Word scno asm("r7") = seL4_SysCall;
  asm volatile ("swi %[swi_num]"
                : "+r" (msg0), "+r" (msg1), "+r" (msg2), "+r" (msg3),
                "+r" (info), "+r" (destptr)
                : [swi_num] "i" __SWINUM(seL4_SysCall), "r"(scno)
                : "memory");

  /* Write out the data back to memory */
  seL4_SetMR(0, msg0);
  seL4_SetMR(1, msg1);
  seL4_SetMR(2, msg2);
  seL4_SetMR(3, msg3);

  return (seL4_MessageInfo_t) {
    .words = {info.words[0]}
  };
}
```

It can be observed that the semantics of this function fall outside the C99 standard, due to its use of non-portable inline assembly and named register hints. It also implicitly relies on operating system support; the `swi` assembly instruction triggers a hardware interrupt that is intended to be handled by the operating system. Currently, we do not attempt to directly interpret these system call stubs, or to represent their precise behaviour. Instead we treat the invocation of a stub as if it itself were the entry point to the kernel.

The semantics we assign to the system calls are implicit, because our model of C currently lacks the expressiveness to describe the effect of the system calls implementing IPC. Specifically, it is not currently possible to directly express the notion of separate virtual address spaces and a function call modifying data in a different address space to that of the caller. Informally, the semantics we assign to the synchronous endpoint invocations that we axiomatise are:

- `seL4_Call` sends the contents of the calling thread's entire IPC buffer, blocks waiting for a corresponding `seL4_Reply` and then receives into the calling thread's IPC buffer;
- `seL4_Reply` unblocks a blocked `seL4_Call` caller and transfers the contents of the IPC buffer of the `seL4_Reply` caller to this receiver; and
- `seL4_Wait` when called on a synchronous endpoint, blocks waiting for a corresponding `seL4_Call` and then receives into the calling thread's IPC buffer.

We axiomatise seL4_ReplyWait as merely a shorthand for seL4_Reply followed by seL4_Wait. That is, we assume its implementation is as follows.

```
seL4_MessageInfo_t seL4_ReplyWait(seL4_CPtr src, seL4_MessageInfo_t msgInfo,
    seL4_Word *sender) {
  seL4_Reply(msgInfo);
  return seL4_Wait(src, sender);
}
```

This definition is faithful to the behaviour of seL4_ReplyWait, though this batching actually occurs within the kernel itself.

For the asynchronous endpoint invocations that we axiomatise, we assign the following semantics:

- seL4_Notify sets the pending bit in an asynchronous endpoint;
- seL4_Poll checks the pending bit in an asynchronous endpoint and clears it; and
- seL4_Wait when called on an asynchronous endpoint, blocks until the pending bit of the endpoint is set and then clears it.

While there is little room for the implementation to deviate from these assumed semantics for the asynchronous endpoint invocations, the semantics we have provided of the synchronous endpoint invocations permit known unfaithful functionality. Our C model cannot currently express distinct virtual memory address spaces, and we do not constrain the extent of the IPC buffer transferred by a system call (usually indicated by the msgInfo parameter). These two properties lead to some unsound possibilities in our formalism. Contrived glue code that accesses regions of the IPC buffer beyond what is transferred can be constructed and proven correct. This is possible because our model represents cooperating threads' IPC buffers as if they were persistent aliases for the same memory. This is a weakness in our formalism, though it is unlikely any user-provided or generated code could exploit this by accident.

Ultimately, it is desirable to show a formal connection between specifications of CAmkES' generated code and a specification of libsel4's code. However, the latter does not currently exist. There have been some previous efforts towards this (Andronick, Klein, and Murray 2012; Boyton 2014) and it seems inevitable that a rich version of such a specification will eventually be created. We have designed our formalism being mindful of this, taking steps to ease a future integration of these two artefacts. However, for now we consider this future work out of the scope of this thesis.

9.2 Verification of More Connector Code

In Chapter 6 we demonstrated generated proofs for the glue code of a specific CAmkES connector, seL4RPCSimple. While the templated proofs we have implemented do not apply to other connectors, it is possible to achieve the same result for another connector by constructing a matching set of templated proofs for its glue code. The seL4RPCSimple connector has been designed with the goal of verification, so we expect verification of another connector that has not had this goal from outset to be more challenging, but still feasible.

Choosing another standard connector, seL4RPCCall, as an example, this connector has similar control flow to seL4RPCSimple but uses calls to memcpy rather than direct variable writes to perform marshalling and unmarshalling. The advantage of this approach is simpler glue code template logic. For example a scalar variable, x, can be marshalled without directly inspecting its type:

```
1 memcpy(buffer, &x, sizeof(x));
```

However, this simplicity becomes a disadvantage when reasoning about the correctness of the resulting memory operations. Reasoning about `memcpy` requires dropping to a lower level memory model in the AutoCorres environment where memory is addressable as a byte array in contrast to the higher level strongly-typed heap. Connecting proofs in one environment to proofs in the other (for example, when dealing with a strongly-typed function that calls `memcpy`) requires explicit embedding and projection functions to be supplied. We have proven a generic correctness lemma of `memcpy` and some lemmas of small functions that call `memcpy` that confirm this approach is feasible. However, we have not attempted to extend this line of work to a full templated proof of the seL4RPCCall connector.

In addition to considering future verification of the standard connectors, we have been mindful of future work to verify user-supplied connectors. There is nothing in our approach that prevents its application to further connectors supplied by the system designer and, in fact, the current CAmkES implementation supports providing templated correctness proofs alongside glue code templates for such connectors. Naturally, the onus is on the supplier of the connector to construct these templated proofs. If the connector is instantiated many times, the proof effort saved by this approach can be significant.

Faced with the prospect of distinct templated proofs for each connector, it is natural to consider whether this can be generalised into a higher level "meta templated" proof. We consider this an open problem, as it is not clear how to abstract over the arbitrary control flow and data manipulation that may be present in glue code. While there are some immediately apparent necessary conditions for wellformed glue code, it is not clear what set of constraints is *sufficient* to guarantee arbitrary glue code correct. If a solution is found to this problem, such a generalisation would be a valuable extension to our work.

9.3 Shared Memory

CAmkES provides an abstraction, dataports, for sharing memory between two component instances. The information flow relationship implied by sharing memory is captured in the seL4 kernel object capability graph, but it is not something we currently handle within the initialisation formalism in Chapter 7. Constructing a CAmkES system that uses shared memory induces an extra relationship in the capability graph that we do not express in the generated access control policy. Thus the resulting generated lemmas that state the component instances cannot affect each other by way of their virtual address space mappings are false, and their generated proofs fail.

The rationale for this limitation is more than simply a proof convenience. The CapDL model does not represent the contents of physical memory. This was a deliberate design decision, as its purpose is to reason purely about kernel objects and capabilities, not data. However, a fundamental consequence of this is that we cannot draw a relationship between CapDL frame objects and a program's code and data. This effectively hampers any reasoning about component instances that share memory.

To understand why this is, consider two component instances with a dataport connection between them. When translated to CapDL, this architecture presents as a collection of objects for each component instance, where capabilities to the same frame object are present in two page tables, one belonging to each component instance. This scenario is depicted in Figure 9.1, where two page tables contain capabilities to `frame2`. This induces an authority graph where the objects of the two component instances are connected by paths involving this frame. Because we cannot, within CapDL, determine the contents of this frame, it is as if these two component instances can affect any aspect of each other's execution. This includes modifying each other's code.

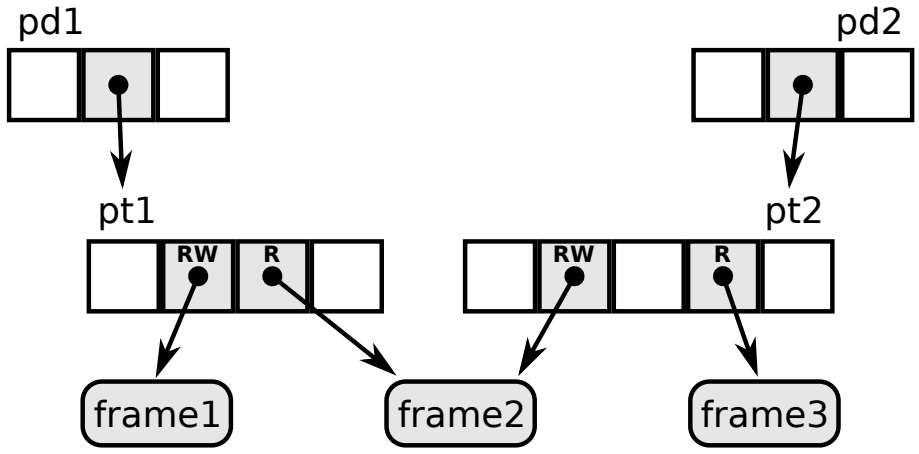

Figure 9.1: Shared memory as represented in CapDL

It is possible to determine a stronger upper bound on the abilities conveyed by shared memory by inspecting component instances' address spaces. All paging configuration, including the offset at which a frame is mapped into a given address space is derivable from a CapDL specification. Thus by combining this knowledge with the regions of each component instance's address space assigned for code and data (extractable from their binaries), it is possible to conclude that a shared page only contains data. This requires the assumption that component instances do not expect to execute code inside a dataport, but this is generally an acceptable limitation.

The primary reason we have not extended our work in this direction is that prior work on correct initialisation of seL4 systems to which we connect does not support allocating shared memory (Boyton 2014). Support for reasoning about a CAmkES system with dataports would require introducing an extra assumption on the way in which systems involving shared memory are initialised. Rather than weaken our existing formalism, we defer this work to a future point at which it can be formally connected to an initialisation guarantee without further assumptions.

9.4 Relaxing Code Requirements

Our approach mandates a specific, rigorous subset of the C language, and some translation steps are deliberately designed to be conservative. A direct consequence of this is that it is possible to write correct code that is not provably correct with our toolchain. This more "risky" code transitively thwarts attempts to prove the correctness of its callers and requires an axiomatised specification to make progress. For example, consider the following contrived function:

```
1  int two(void) {
2    int x = 0;
3    return (++x&&x++,x++);
4  }
```

Though this is valid C99 code, it is not accepted by our toolchain. It is clear how to rephrase this trivial function into returning a literal value, but not all unsupported code constructs can be so easily adapted. However, it should be obvious that this example does not represent clear and concise code. It is our position that the majority of unsupported code can be characterised this way.

We claim that many of the constructs used to write such code are inherently unsafe for use in high-assurance environments. Constructs that are not supported by our toolchain have not been arbitrarily excluded. Rather, deliberate decisions were made to avoid language features that are unnecessary and increase the probability of incorrect code. The remaining subset has been shown sufficient to implement both an operating system kernel and a fully functional component platform, demonstrating that these limitations do not prevent the implementation of complex systems. To put the argument against the use of unprovable constructs in simple terms, if you can't formally prove the correctness of your code, why would you entrust it with your life?

9.5 Concurrency

It is well known that writing correct concurrent code is challenging and techniques for verifying concurrent code are still relatively primitive. Driven by implicit assumptions in our toolchain, our present work assumes that all component instance code executes in a single threaded environment. This assumption holds true in many complex CAmkES systems, and high-assurance systems are generally designed to be single-threaded where possible. From here, it is possible to rely on the isolation properties of seL4 to prevent interference between component instances.

However, this guarantee relies on informal arguments and, moreover, performance concerns often motivate the development of multithreaded components. A partial roadmap for extending our work to concurrent settings emerges by observing that it is not merely coincident, but actually *required* that CAmkES' generated code runs without interference. The glue code operates on state that, for correctness, must be thread-local. This relationship is informally observable statically in the current implementation. That is, the data accesses performed by glue code are observably safe, if we assume all hand written component code avoids accessing the glue code's private state. Thus it should be possible to extend our work to a concurrent setting by way of an additional assumption that the user-provided code only interacts with the glue code in a wellformed manner. However, our current C semantics are lacking the expressiveness to capture this property and this would require a more rich concept of execution. Therefore, we leave this for future work.

9.6 Incremental Verification

The work of this thesis is primarily targeted at systems that are already componentised. While CBSE is a widely deployed system framework in high-assurance software, it is not ubiquitous. Leveraging our approach requires the use of CAmkES, or another component platform with equivalent guarantees, and therefore any existing monolithic system would need to be first decomposed into components and connections prior to verification. For large systems, it is desirable to do this incrementally to smooth the transition.

From an engineering perspective, a natural way to transition a monolithic (often Linux-based) system to a componentised, microkernel-based environment is to first virtualise the existing system as a component (Andronick, Greenaway, and Elphinstone 2010; Heiser et al. 2013). From there, selected functionality can be extracted into native components, step-by-step. At any point in this process, the verification techniques we have shown would provide guarantees on only the native components and their connections. That is, functionality still inside the monolithic virtualised system would remain opaque from a system reasoning standpoint. This is both a blessing and a curse. Verifying such a system that relies on critical functionality inside this monolithic system is most likely intractable. However, conversely, the only functionality that need be extracted from the monolithic system is the minimal set required to prove correctness. That is, non-critical system functionality can be safely incorporated into a component whose code is never analysed. By leveraging

the isolation implied by the system architecture, this code need not be an input to verification. Consequently, this can enable verification to scale to larger code bases, and increases the utility of CBSE aside from purely engineering motivations.

We are aware of no fundamental limitation that would prevent the applicability of the approach described, but currently such a translation would be entirely manual. There exists no tooling or automation for migrating monolithic systems to a CAmkES- and seL4-based environment. While this would be a natural extension to our work, we have not pursued this as it has little novelty from a research standpoint.

9.7 Architectural Patterns

Treating component-based systems from a higher level perspective, correctness criteria for a particular system may involve conformance to a particular safety or security pattern (Yoder and Barcalow 1997; Yoshioka, Washizaki, and Maruyama 2008). With the fine-grained access control policy obtainable from the formalism in Chapter 7, it is possible to show how a component-based system implements a particular pattern. This approach can be extended to characterising the feature set of capability descriptions that satisfy a particular pattern (Rimba 2013; Rimba et al. 2015).

Such work constitutes a next "level" of reasoning about the correctness of a component-based system. An important precondition for demonstrating the correct implementation of a pattern, is the correctness of the underlying mechanisms on which a system is built; the target of our present work. We see our efforts as freeing designers from thinking about the low level details of their system and allowing them to reason at a more intuitive level. Much like a higher level language serves as a productivity multiplier for a programmer, we view our work as extending the power and scope of verification efforts. This higher reasoning is a logical next step for verification of component-based systems.

9.8 Beyond Functional Correctness

The work presented provides support for proving functional correctness of a component-based system. In certain domains, functional correctness is only part of the correctness criteria. For example, realtime systems require, in addition to functional correctness, guarantees about the timely execution of code. An aircraft collision avoidance system is considered incorrect if it delivers suitable guidance to avoid an impact, but the guidance is only provided after the impact has occurred. We do not attempt to solve such problems with our current work, though our approach does not preclude any future integration with techniques for showing properties beyond functional correctness. Our work has been conducted mindful of other ongoing seL4-based assurance efforts (Cock et al. 2014; Lyons and Heiser 2014; Sewell, Myreen, and Klein 2013) such that integration with orthogonal techniques should prove straightforward.

9.9 Erroneous Specifications

There is a fundamental limitation in software verification that it is only ever possible to prove the property you state. In much the same way as a correct program can only ever do the things its code instructs, a verification effort is inherently bounded by the specification provided. An unavoidable consequence is that an incorrect system can be proven "correct" according to an incorrect specification. Put another way, a proven system is only ever as correct as its specification.

Specification errors, while not the most prevalent cause of software faults, do occur in real world high-assurance systems. The Ariane 5 rocket suffered a catastrophic failure moments after its launch, due to a fault later partially attributed to an error in the specification of its Inertial Reference System (SRI) (Lions 1996):

> *The specification of the exception-handling mechanism also contributed to the failure. In the event of any kind of exception, the system specification stated that: the failure should be indicated on the databus, the failure context should be stored in an EEPROM memory (which was recovered and read out for Ariane 501), and finally, the SRI processor should be shut down.*
>
> *It was the decision to cease the processor operation which finally proved fatal.*

In this system, it was *specified* behaviour that the SRI processor should shut down on exception, though it became clear after this incident that this was incorrect behaviour. A verification effort intended to show correspondence to a specification can never detect errors of this nature.

Our work was not intended to target this problem and it does not help safeguard the system designer against this class of errors. However, it is important to acknowledge that this issue exists. While we have improved upon the available techniques for designers to prove the correctness of their system, our approach still admits a correctness proof against a faulty specification. We provide you with a more expressive language for correctness, but you still need to know what you want to say.

9.10 Summary

The techniques we have introduced in this thesis have real world applicability and utility and advance the state of the art in specific areas, but they are imperfect. In this chapter, we have detailed the bounds to our current work and how it could be extended in future. In the process of performing this research, we have come to recognise the potential for verification techniques to revolutionise the way software is built. While our work represents a step forward in software verification for high-assurance systems, we see it as one of many necessary advances on the path to truly reliable correct software.

Chapter 10

Conclusion

No one sets out to build an incorrect program. Errors in software systems and the behaviours we refer to as "bugs" are inevitably the result of mistakes, whether through ignorance, carelessness or simply accident. Finding bugs through testing and attributing their root causes is challenging and often relies more on luck than a reliable process. As the scale and scope of software increases, so too does the impact of its bugs. In safety- and security-critical software, the consequences of errors are often unacceptable, yet these systems are still constructed using approaches that only provide weak guarantees.

It is our position that it is impossible to repeatedly build truly reliable software systems without a process that assigns a rigorous semantics to programs and provides strong correctness guarantees. The strongest machinery we have for building correct, safe and secure software is formal verification. Yet it has proven challenging for large engineering projects to apply.

In this thesis, we have demonstrated how to decrease the barriers to formal verification by architectural decomposition and pervasive automation. By providing techniques well aligned with the componentisation of a system, we give a system designer significant control over the form and complexity of proof obligations through their architectural description. The parallels between system architecture and verification efforts aid the intuition of the system designer, and bring formal verification closer to the engineering domain they are most likely more familiar with.

10.1 Summary of Formalisms

We have presented three techniques for verification of a system of components, each situated at a different point on a spectrum between automation and expressiveness:

- A static architecture model, capable of describing a system's component instances and their connections as they exist at design time (Chapter 4);
- A dynamic behavioural model, allowing expression of the runtime execution effects of a set of component instances and connections (Chapter 5); and
- A code-level model for functional correctness, enabling partially-automated interactive verification of component instance code (Chapter 6).

For each of these, we have implemented automatic generation of base definitions from the architectural description of any given component-based system. We also automate all representations of code generated by

the component platform itself, and proof obligations relating to the correctness of this code. The final environment we achieve provides the system designer with the ability to reason in their chosen model, having only to deal with the hand written code and behaviour of their component instances.

To verify higher level abstract properties of a component-based system, we provide the designer with a single formalism:

- A generated capability specification and access control policy for reasoning about information flow at the granularity of component instances and connections (Chapter 7).

Finally, to obviate the need for the designer to trust the component platform itself, we verify its correctness in two complementary efforts:

- Automatic generation of code-level proofs for the generated code of an RPC connector (Chapter 6); and
- A formal definition of the component platform's logic for generating kernel objects and capabilities, and a proof that the accompanying generated access control policy abstracts this capability specification (Chapter 7).

The practical outcomes of these efforts for the system designer, are less effort and a lower cognitive load. Rather than concerning themselves with the correctness of all the code within their system, they need only consider the code for their component instances.

10.2 Contributions

Our over-arching aim in this research has been enabling formal verification at greater scale and pursuing generated proofs for generated code. To that end, our contributions in this thesis have been fourfold:

- Component platform correctness. The automated proofs we have demonstrated provide a strong guarantee of the correctness of the CAmkES component platform. To our knowledge, CAmkES is currently the only code-level verified component platform in existence. Our techniques generalise and our approach could be applied to another static component platform in the future.
- Automated verification of generated code. We have demonstrated this for a specific CAmkES RPC connector and detailed how this work could be extended to the verification of further connector code. This work is exemplary of a broader approach for automated verification of generated code by exploiting properties of its known regularity.
- A methodology for scaling software verification. The techniques we have introduced provide a framework for managing complexity and partitioning verification. By doing so, we enable verification of larger code bases than was previously possible and reduce the manual effort required.
- A path to whole system assurance. The verification of large systems has previously been impeded by difficulty in reasoning across system module boundaries. Our work demonstrates how to overcome this challenge and how to verify a cooperative system of hand written and generated code.

With these contributions, we aim to further the reach of formal verification and increase its attractiveness to high-assurance system practitioners.

10.3 The Road Ahead

When a software system is put in charge of our secrets, our safety or our health, it should earn the right to do so. Formal verification can significantly lower defect rates in high-assurance software, and is the best technique we have available to validate a system's worthiness in this respect. On the path to ubiquitous, formally verified, high-assurance software we see our work as a small but significant step. We optimistically look forward to a world in which you never have to wonder, "is it correct?"

List of Figures

References

1. Allen, Robert. 1996. "HLA: a Standards Effort as Architectural Style." In *Proceedings of the 2nd International Software Architecture Workshop*. San Francisco, CA, USA.

2. Allen, Robert, and David Garlan. 1997. "A Formal Basis for Architectural Connection." *ACM Transactions on Software Engineering and Methodology* 6 (3) (July): 213–249.

3. Alves-Foss, Jim, Paul W. Oman, Carol Taylor, and Scott Harrison. 2006. "The MILS Architecture for High-Assurance Embedded Systems." *International Journal on Embedded Systems* 2: 239–247.

4. Andronick, June, David Greenaway, and Kevin Elphinstone. 2010. "Towards Proving Security in the Presence of Large Untrusted Components." In *Systems Software Verification*, edited by Ralf Huuck, Gerwin Klein, and Bastian Schlich. Vancouver, Canada.

5. Andronick, June, Gerwin Klein, and Toby Murray. 2012. "Formal System Verification for Trustworthy Embedded Systems, Final Report for AOARD Grant #FA2386-10-1-4105." Sydney, Australia: NICTA.

6. Andronick, June, Corey Lewis, and Carroll Morgan. 2015. "Controlled Owicki-Gries Concurrency: Reasoning About the Preemptible eChronos Embedded Operating System." In *Workshop on Models for Formal Analysis of Real Systems (MARS 2015)*. Suva, Fiji.

7. *AUTOSAR Release 4.2 Overview and Revision History*. 2015. AUTOSAR; http://www.autosar.org/fileadmin/files/releases/4-2/AUTOSAR_TR_ReleaseOverviewAndRevHistory.pdf.

8. Back, Ralph-Johan. 1978. "On the Correctness of Refinement Steps in Program Development." PhD thesis, Helsinki, Finland: University of Helsinki.

9. Ballarin, Clemens. 2010. "Tutorial to Locales and Locale Interpretation." In *Contribuciones Científicas En Honor de Mirian Andrés*, edited by Laureano Lambán, Ana Romero, and Julio Rubio. Logroño, Spain.

10. Basu, Ananda, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. 2011. "Rigorous Component-Based System Design Using the BIP Framework." *IEEE Software* 28 (3): 41–48.

11. Basu, Ananda, Laurent Mounier, Marc Poulhiès, Jacques Pulou, and Joseph Sifakis. 2007. "Using BIP for Modeling and Verification of Networked Systems - a Case Study on TinyOS-Based Networks." In *Proceedings of the 6th IEEE Symposium on Network Computing and Applications*, 257–260. Cambridge, MA, USA.

12. Batory, Don, and Bart J. Geraci. 1997. "Composition Validity and Subjectivity in GenVoca Generators." *IEEE Transactions on Software Engineering* 23 (2) (February): 67–82.

13. Blundell, Colin, Dimitra Giannakopoulou, and Corina S. Păsăreanu. 2006. "Assume-Guarantee Testing." *ACM SIGSOFT Software Engineering Notes* 31 (2) (March): 1–8.

14. Boebert, W.E. 1984. “On the Inability of an Unmodified Capability Machine to Enforce the ⋆–Property.” In *Proceedings of the 7th DoD/NBS Computer Security Conference*, 291–293.

15. Böhme, Sascha, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. 2010. “HOL-Boogie — an Interactive Prover-Backend for the Verifying C Compiler.” *Journal of Automated Reasoning* 44 (1–2) (February): 111–144.

16. Boyton, Andrew. 2014. “Secure Architectures on a Verified Microkernel.” PhD thesis, Sydney, Australia: CSE, UNSW.

17. Boyton, Andrew, June Andronick, Callum Bannister, Matthew Fernandez, Xin Gao, David Greenaway, Gerwin Klein, Corey Lewis, and Thomas Sewell. 2013. “Formally Verified System Initialisation.” In *International Conference on Formal Engineering Methods*, edited by Jing Sun Lindsay Groves, 70–85. Queenstown, New Zealand.

18. Boyton, Andrew, Simon Rodgers, Matthew Fernandez, Cyril Bur, and Adrian Danis. 2014. “CapDL Initialiser+loader, Version 1.0.0.” https://github.com/seL4/CapDL.

19. Brinch Hansen, Per. 1970. “The Nucleus of a Multiprogramming Operating System.” *Communications of the ACM* 13: 238–250.

20. Burstein, Mark, Drew McDermott, Douglas R. Smith, and Stephen J. Westfold. 2003. “Derivation of Glue Code for Agent Interoperation.” *Journal of Autonomous and Multi-Agent Systems* 6: 265–286.

21. CBS News. 2010. “Toyota ‘Unintended Acceleration’ Has Killed 89.”

22. Cervantes, Humberto, and Richard S. Hall. 2004. “Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model.” In *Proceedings of the 26th International Conference on Software Engineering*, 614–623. Edinburgh, Scotland.

23. Chander, Ajay, Drew Dean, and John C. Mitchell. 2001. “A State-Transition Model of Trust Management and Access Control.” In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, 27–43. Cape Breton, Nova Scotia, Canada.

24. Cobleigh, Jamieson M., Dimitra Giannakopoulou, and Corina S. Păsăreanu. 2003. “Learning Assumptions for Compositional Verification.” In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 331–346. Warsaw, Poland.

25. Cock, David, Qian Ge, Toby Murray, and Gernot Heiser. 2014. “The Last Mile: an Empirical Study of Some Timing Channels on seL4.” In *ACM Conference on Computer and Communications Security*, 570–581. Scottsdale, AZ, USA.

26. Cock, David, Gerwin Klein, and Thomas Sewell. 2008. “Secure Microkernels, State Monads and Scalable Refinement.” In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, edited by Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, 167–182. Montreal, Canada.

27. Common Criteria Recognition Arrangement. 2012. “Common Criteria for Information Technology Security Evaluation: Part 3: Security Assurance Components V3.1r4.”

28. *Common Object Request Broker Architecture 3.3 Specification*. 2012. Object Management Group, Inc. http://www.omg.org/spec/CORBA/3.3/.

29. *Common Object Request Broker Architecture Components*. 1999. Object Management Group, Inc.

30. *Common Object Request Broker Architecture Specification, Version 1.0*. 1991. Object Management Group, Inc. http://www.omg.org/spec/CORBA/1.0/.

31. Crnkovic, Ivica. 2001. “Component-Based Software Engineering - New Challenges in Software Development.” *Software Focus* 2 (4): 127–133.

32. Delaware, Benjamin, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. “Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant.” In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Mumbai, India.

33. Denning, Dorothy. E. 1976. “A Lattice Model of Secure Information Flow.” *Communications of the ACM* 19: 236–242.

34. Dennis, Jack B., and Earl C. Van Horn. 1966. “Programming Semantics for Multiprogrammed Computations.” *Communications of the ACM* 9: 143–155.

35. Dietz, Will, Peng Li, John Regehr, and Vikram Adve. 2012. “Understanding Integer Overflow in C/C++.” In *Proceedings of the 34th International Conference on Software Engineering*, 760–770. Piscataway, NJ, USA.

36. Engler, Dawson R., Benjamin Chelf, Andy Chou, and Seth Hallem. 2000. “Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions.” In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 1–16. San Diego, CA, USA.

37. Fassino, Jean-Philippe, Jean-Bernard Stefani, Julia Lawall, and Gilles Muller. 2002. “THINK: a Software Framework for Component-Based Operating System Kernels.” In *Proceedings of the 2002 USENIX Annual Technical Conference*, 73–86. Monterey, CA, USA.

38. Feiertag, Richard J., and Peter G. Neumann. 1979. “The Foundations of a Provably Secure Operating System (PSOS).” In *AFIPS Conference Proceedings, 1979 National Computer Conference*, 329–334. New York, NY, USA.

39. Fernandez, Matthew, June Andronick, Gerwin Klein, and Ihor Kuz. 2015a. “Automated Verification of RPC Stub Code.” In *International Symposium on Formal Methods*, 273–290. Oslo, Norway.

40. Fernandez, Matthew, June Andronick, Gerwin Klein, and Ihor Kuz. 2015b. “Automated Verification of a Component Platform.” Sydney, Australia: NICTA; UNSW.

41. Fernandez, Matthew, Peter Gammie, June Andronick, Gerwin Klein, and Ihor Kuz. 2013. “CAmkES Glue Code Semantics.” Sydney, Australia: NICTA; UNSW.

42. Fernandez, Matthew, Gerwin Klein, Ihor Kuz, and Toby Murray. 2013. “CAmkES Formalisation of a Component Platform.” Sydney, Australia: NICTA; UNSW.

43. Fernandez, Matthew, Ihor Kuz, Gerwin Klein, and June Andronick. 2013. “Towards a Verified Component Platform.” In *Workshop on Programming Languages and Operating Systems (PLOS)*, 6. Farmington, PA, USA.

44. Feske, Norman, and Christian Helmuth. 2004. “Overlay Window Management: User Interaction with Multiple Security Domains.” TUD-FI04-02-März-2004. Technische Universität Dresden.

45. Feske, Norman, and Christian Helmuth. 2005. “A Nitpicker’s Guide to a Minimal-Complexity Secure GUI.” In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, 85–94. Washington, DC, USA.

46. Feske, Norman, and Christian Helmuth. 2007. “Design of the Bastei OS Architecture.” TUD-FI06-07-Dezember-2006. Technische Universität Dresden.

47. Flatt, Matthew, and Matthias Felleisen. 1998. “Units: Cool Modules for HOT Languages.” In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 236–248. Montreal, Canada.

48. Ford, Bryan, Godmar Back, Greg Benson, Jay Leprau, Albert Lin, and Olin Shivers. 1997. “The Flux OSKit: a Substrate for Kernel and Language Research.” In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 38–51. St Malo, France.

49. Formaggio, Yannick. 2015. “Attacking VxWorks: from Stone Age to Interstellar.” In *44CON Cyber Security*. London, UK.

50. Gabber, Eran, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. 1999. “The Pebble Component-Based Operating System.” In *Proceedings of the 1999 USENIX Annual Technical Conference*, 267–282. Monterey, CA, USA.

51. Gammie, Peter. 2015. “Concurrent IMP.” *Archive of Formal Proofs* (April).

52. Garlan, David, Robert Allen, and John Ockerbloom. 1995. “Architectural Mismatch or Why It’s Hard to Build Systems Out of Existing Parts.” In *Proceedings of the 17th International Conference on Software Engineering*, 179–185. Seattle, WA, USA.

53. Gay, David, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. “The nesC Language: a Holistic Approach to Networked Embedded Systems.” In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1–11. San Diego, CA, USA.

54. Genßler, Thomas, and Christian Zeidler. 2001. “Rule-Driven Component Composition for Embedded Systems.” In *Proceedings of the 23rd International Conference on Software Engineering: Workshop on Component-Based Software Engineering*. Toronto, Canada.

55. Genßler, Thomas, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller, and Chris Stich. 2002. “Components for Embedded Software: the PECOS Approach.” In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 19–26. Grenoble, France.

56. Giannakopoulou, Dimitra, Corina S. Păsăreanu, and Howard Barringer. 2002. “Assumption Generation for Software Component Verification.” In *Proceedings of the 17th IEEE/ACM International Conference on Automated Software Engineering*, 3–12. Edinburgh, UK.

57. Greenaway, David. 2015. “Automated Proof-Producing Abstraction of C Code.” PhD thesis, Sydney, Australia: CSE, UNSW.

58. Hallem, Seth, Benjamin Chelf, Yichen Xie, and Dawson Engler. 2002. “A System and Language for Building System-Specific, Static Analyses.” In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 69–82. Berlin, Germany.

59. Hardy, Norman. 1985. “KeyKOS Architecture.” *ACM Operating Systems Review* 19 (4) (October): 8–25.

60. Hawkins, Peter, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. 2011. “Data Representation Synthesis.” In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 38–49. San Jose, CA, USA.

61. Hayden, Mark. 1998. “The Ensemble System.” PhD thesis, Ithaca, NY, USA: Cornell University.

62. Heiser, Gernot, Etienne Le Sueur, Adrian Danis, Aleksander Budzynowski, Tudor-Ioan Salomie, and Gustavo Alonso. 2013. “RapiLog: Reducing System Complexity Through Verification.” In *EuroSys Conference*, 323–336. Prague, Czech Republic.

63. Helander, Johannes, and Alessandro Forin. 1998. “MMLite: a Highly Componentized System Architecture.” In *Proceedings of the 8th SIGOPS European Workshop*. Sintra, Portugal.

64. Hieb, Jeffrey, and James Graham. 2009. “Designing Security-Hardened Microkernels for Field Devices.” In *Critical Infrastructure Protection II*, edited by M. Papa and S. Shenoi, 290/2009:129–140. IFIP International Federation for Information Processing. Springer.

65. Hoare, C. A. R. 1985. *Communicating Sequential Processes*. Prentice Hall.

66. *IEC 61508: Functional Safety of Electrical, Electronic and Programmable Electronic Safety Related Systems*. 1998. International Electrotechnical Commission.

67. ISO/IEC. 2005. “Programming Languages — C.” 9899:TC2. ISO/IEC JTC1/SC22/WG14.

68. *ISO26262: Road Vehicles – Functional Safety*. 2011. ISO.

69. Jones, C. B. 1983. “Tentative Steps Towards a Development Method for Interfering Programs.” *ACM Transactions on Programming Languages and Systems* 5 (4): 596–619.

70. Klein, Gerwin, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, et al. 2010. “SeL4: Formal Verification of an Operating-System Kernel.” *Communications of the ACM* 53 (6) (June): 107–115.

71. Klein, Gerwin, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. “Comprehensive Formal Verification of an OS Microkernel.” *ACM Transactions on Computer Systems* 32 (1) (February): 2:1–2:70.

72. Klein, Gerwin, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, et al. 2009. “SeL4: Formal Verification of an OS Kernel.” In *ACM Symposium on Operating Systems Principles*, 207–220. Big Sky, MT, USA: ACM.

73. Kreitz, Christoph. 1997. “Formal Reasoning About Communication Systems I: Embedding ML into Type Theory.” Ithaca, NY, USA: Cornell University.

74. Kuz, Ihor, Gerwin Klein, Corey Lewis, and Adam Christopher Walker. 2010. “CapDL: a Language for Describing Capability-Based Systems.” In *Asia-Pacific Workshop on Systems (APSys)*, 31–35. New Delhi, India.

75. Kuz, Ihor, Yan Liu, Ian Gorton, and Gernot Heiser. 2007. “CAmkES: a Component Model for Secure Microkernel-Based Embedded Systems.” *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems* 80 (5) (May): 687–699.

76. Laroux, Paul, and Bill Graham. 2009. “Secure by Design: Using a Microkernel RTOS to Build Secure, Fault-Tolerant Systems.” QNX.

77. Leroy, Xavier. 2006. “Formal Certification of a Compiler Back-End, or: Programming a Compiler with a Proof Assistant.” In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, edited by J. G. Morrisett and S. L. P. Jones, 42–54. Charleston, SC, USA.

78. Levis, Philip, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, et al. 2005. “TinyOS: an Operating System for Sensor Networks.” In *Ambient Intelligence*, edited by Werner Weber, Jan Rabaey, and Emile H. L. Aarts, 115–148. Springer Verlag.

79. Levy, Henry M. 1984. *Capability-Based Computer Systems*. Digital Press.

80. Lewis, Corey, Gerwin Klein, Andrew Boyton, David Greenaway, and Matthew Fernandez. 2014. “CapDL Tool Set, Version 1.0.0.” https://github.com/seL4/CapDL.

81. Liedtke, Jochen. 1993. “Improving IPC by Kernel Design.” In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 175–188. Asheville, NC, USA.

82. Liedtke, Jochen. 1995. “On μ-Kernel Construction.” In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 237–250. Copper Mountain, CO, USA.

83. Lions, J. L. 1996. “Ariane 5 Flight 501 Failure: Report by the Inquiry Board.” https://www.ima.umn.edu/~arnold/disasters/ariane5rep.html.

84. Liu, Xiaoming, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. 1999. “Building Reliable, High-Performance Communication Systems from Components.” In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 80–92. Charleston, SC, USA.

85. Lyons, Anna, and Gernot Heiser. 2014. “Mixed-Criticality Support in a High-Assurance, General-Purpose Microkernel.” In *Workshop on Mixed Criticality Systems*, edited by Rob Davis and Liliana Cucu-Grosjean, 9–14. Rome, Italy.

86. McDonald, James, and John Anton. 2001. “Specware: Producing Software Correct by Construction.” Palo Alto, CA, USA: Kestrel Institute.

87. Meyer, Bertrand. 1997. “The Next Software Breakthrough.” *IEEE Computer* 30 (7) (July): 113–114.

88. Meyer, Bertrand. 2003. “The Grand Challenge of Trusted Components.” In *Proceedings of the 25th International Conference on Software Engineering*. Portland, OR, USA.

89. “Microvisor Products - General Dynamics Mission Systems.” 2016. https://gdmissionsystems.com/cyber/products/trusted-computing-cross-domain/microvisor-products/.

90. Miller, Mark, Ka-Ping Yee, and Jonathan Shapiro. 2003. “Capability Myths Demolished.” HP Labs.

91. Mitchell, James G., Jonathan J. Gibbons, Graham Hamilton, Peter B. Kessler, Yousef A. Khalidi, Panos Kougiouris, Peter W. Madany, Michael N. Nelson, Michael L. Powell, and Sanjay R. Radia. 1994. “An Overview of the Spring System.” In *Spring COMPCON*, 122–131. San Francisco, CA, USA.

92. Morgan, Carroll. 1990. *Programming from Specifications*. 2nd ed. Prentice Hall.

93. Müller, Peter, Christian Zeidler, Christian Stich, and Andreas Stelter. 2001. “PECOS - Pervasive Component Systems.” In *Workshop on Open Source Technologie in Der Automatisierungstechnik, GMA Kongress*. Baden-Baden, Germany.

94. Murray, Toby, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. 2012. “Noninterference for Operating System Kernels.” In *International Conference on Certified Programs and Proofs*, edited by Chris Hawblitzel and Dale Miller, 126–142. Kyoto, Japan: Springer.

95. Murray, Toby, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. “SeL4: from General Purpose to a Proof of Information Flow Enforcement.” In *IEEE Symposium on Security and Privacy*, 415–429. San Francisco, CA, USA.

96. National Highway Traffic and Safety Administration. 2016. “Federal Motor Vehicle Safety Standards Standard No. 208: Occupant Crash Protection.”

97. National Security Agency, Systems, Information Assurance Directorate, and Network Analysis Center. 2010. “Separation Kernels on Commodity Workstations.” http://www.niap-ccevs.org/announcements/Separation%20Kernels%20on%20Commodity%20Workstations.pdf.

98. Necula, George C. 1997. “Proof-Carrying Code.” In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 106–119. Paris, France.

99. Necula, George C. 2000. “Translation Validation for an Optimizing Compiler.” In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 83–94. Vancouver, British Columbia, Canada.

100. Nipkow, Tobias, Lawrence Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — a Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer.

101. Norrish, Michael. 1998. “C Formalised in HOL.” PhD thesis, Cambridge, UK: University of Cambridge Computer Laboratory.

102. Ommering, Rob van, Frank van der Linden, Jeff Kramer, and Jeff Magee. 2000. “The Koala Component Model for Consumer Electronics Software.” *IEEE Computer* 33 (3) (March): 78–85.

103. *OSEK/VDX Operating System Specification, Version 2.2.3*. 2005. Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen/Vehicle Distributed Executive; http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

104. Owicki, Susan, and David Gries. 1976. “An Axiomatic Proof Technique for Parallel Programs.” *Acta Informatica* 6: 319–340.

105. Parrish, Patricia G. 2013. “Bookout V. Toyota Motor Corporation.” District Court, Oklahoma County, OK, USA.

106. Pnueli, Amir, Michael Siegel, and Eli Singerman. 1998. “Translation Validation.” In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 151–166. Lisbon, Portugal.

107. Reid, Alastair, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. 2000. “Knit: Component Composition for Systems Software.” In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 347–360. San Diego, CA, USA.

108. Reynolds, John C. 2002. “Separation Logic: a Logic for Mutable Data Structures.” In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, 55–74. Copenhagen, Denmark.

109. Rimba, Paul. 2013. “Building High Assurance Secure Applications Using Security Patterns for Capability-Based Platforms.” In *International Conference on Software Engineering*, 4. San Francisco, CA, USA.

110. Rimba, Paul, Liming Zhu, Xiwei (Sherry) Xu, and Wei (Daniel) Sun. 2015. “Building Secure Applications Using Pattern-Based Design Fragments.” In *The Second International Workshop on Dependability and Security of System Operation*, 19–24. Montreal, Canada.

111. Rushby, John. 1981. “Design and Verification of Secure Systems.” In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, 12–21. Pacific Grove, CA, USA.

112. Rushby, John. 1984. “A Trusted Computing Base for Embedded Systems.” In *Proceedings of the 7th DoD/NBS Computer Security Conference*, 294–311.

113. Saltzer, Jerome H. 1974. “Protection and the Control of Information Sharing in Multics.” *Communications of the ACM* 17: 388–402.

114. Sewell, Thomas, Magnus Myreen, and Gerwin Klein. 2013. “Translation Validation for a Verified OS Kernel.” In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 471–481. Seattle, WA, USA: ACM.

115. Sewell, Thomas, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. 2011. “SeL4 Enforces Integrity.” In *International Conference on Interactive Theorem Proving*, edited by Marko

van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, 325–340. Nijmegen, The Netherlands: Springer.

116. Sexton, Darren. 2013. "An Outline Workflow for Practical Formal Verification from Software Requirements to Object Code." In *Proceedings of the 18th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, 32–47. Madrid, Spain.

117. Shapiro, J.S. 1997. "EROS: a Capability System." MS-CIS-97-04. Department of Computer; Information Science, University of Pennsylvania.

118. Shapiro, Jonathan S. 1999. "EROS: a Capability System." PhD thesis, Philadelphia, PA, USA: University of Pennsylvania.

119. Shapiro, Jonathan S. 2003. "Thinking About Capabilities: an EROS-Centric View." John Hopkins University; university presentation.

120. Shapiro, Jonathan S., Jonathan M. Smith, and David J. Farber. 1999. "EROS: a Fast Capability System." In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, 170–185. Charleston, SC, USA.

121. Silva, Leandro Dias da, Kyller Gorgônio, and Angelo Perkusich. 2008. "Petri Nets for Component-Based Software Development." In *Petri Net, Theory and Applications*, edited by Vedran Kordic, 471–496. Vienna, Austria: I-Tech Education; Publishing.

122. Szyperski, Clemens. 1997. *Component Software: Beyond Object-Oriented Programming*. Essex, England: Addison-Wesley/ACM Press.

123. Transportation Research Board. 2012. "The Safety Promise and Challenge of Automotive Electronics." Edited by National Research Council of the National Academies. National Academy of Sciences.

124. Trustworthy Systems Team. 2014a. "SeL4 V1.03, Release 2014-08-10." https://github.com/seL4/seL4.

125. Trustworthy Systems Team. 2014b. "SeL4 Proofs for API 1.03, Release 2014-08-10." https://github.com/seL4/l4v.

126. Trustworthy Systems Team. 2015. "CAmkES Release 2.0.0." https://github.com/seL4/CAmkES.

127. Varda, Kenton. 2008. "Protocol Buffers." https://developers.google.com/protocol-buffers/.

128. Varda, Kenton. 2013. "Cap'n Proto." https://capnproto.org/.

129. Vogt, Dirk, Björn Döbel, and Adam Lackorzynski. 2010. "Stay Strong, Stay Safe - Enhancing Reliability of a Secure Operating System." In *1st Workshop on Isolation and Integration for Dependable Systems*. Paris, France.

130. Wang, Xi, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. "Undefined Behavior: What Happened to My Code?" In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*, 9:1–9:7. New York, NY, US.

131. Wang, Xi, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. "Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behaviour." In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 260–275. Farmington, PA, USA.

132. Wenzel, Makarius. 2014. *The Isabelle/Isar Reference Manual*.

133. Winwood, Simon, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. 2009. "Mind the Gap: a Verification Framework for Low-Level C." In *International Conference on Theorem Proving in Higher Order Logics*, edited by S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, 500–515. Munich, Germany.

134. Woods, John F. 1992. “Why Is This Legal?” https://groups.google.com/forum/?hl=en#!msg/comp.std.c/ycpVKxTZkgw/S2hHdTbv4d8J.

135. Wulf, William, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, C. Pierson, and Fred Pollack. 1974. “HYDRA: the Kernel of a Multiprocessor Operating System.” *Communications of the ACM* 17: 337–345.

136. Yellin, Daniel M., and Robert E. Strom. 1994. “Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors.” In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 176–190. Portland, OR, USA.

137. Yellin, Daniel M., and Robert E. Strom. 1997. “Protocol Specifications and Component Adaptors.” *ACM Transactions on Programming Languages and Systems* 19 (2) (March): 292–333.

138. Yoder, Joseph, and Jeffrey Barcalow. 1997. “Architectural Patterns for Enabling Application Security.” In *Proceedings of the 4th Pattern Languages of Programs*, 301–336. Monticello, IL, USA.

139. Yoshioka, Nobukazu, Hironori Washizaki, and Katsuhisa Maruyama. 2008. “A Survey on Security Patterns.” *Progress in Informatics* 1 (5) (March): 35–47.

Appendix A

Composite Components as Syntactic Sugar

CAmkES provides the ability to define "composite components" that are components that contain other components. The utility of this feature, which is present in several component platforms, is to ease coarser granularity reuse of components. For example, a portable component that describes a complex piece of hardware can be defined as a component containing a subcomponent for each functional unit of the hardware device.

Composite components are not dealt with directly by the formalisms in this thesis, but we claim this is not a weakness as composite components are a form of syntactic sugar. In the following, we demonstrate how to desugar a composite component definition.

Consider the specification on the following page as an example of an architecture description using composite components. In this specification, the component `Composite` is a composite component type containing two component instances, `p1` and `p2`, of the same type. The `composition` block of a composite component definition can contain instantiations of components and connectors (similar to the `composition` block of an `assembly` block), as well as `export` statements that give internal interfaces, externally visible names. In this specification, the `export` statements make `Composite`'s interfaces, `i` and `o`, aliases for its contained components' interfaces, `p1.i` and `p2.o`, respectively. Observe that the composite component type can have its own, distinct interfaces (lines 20-21) and can be instantiated multiple times (lines 41-42), just like any other component type. This architecture is depicted in Figure A.1.

To redefine this specification without using composite components, we would merge its `composition` block into the `assembly` block's `composition`, and expand its statements. The modified specification appears on page 165. Note that the component instances that were defined within `Composite`'s definition now appear in the final `composition` (lines 33-37). Similarly, the connection that was defined with `Composite`'s definition is now expanded in the final `composition` (lines 45-47). This expanded architecture is depicted in Figure A.2.

Comparing this desugared specification to the original specification, the utility of composite components becomes clear. The original specification is much more concise and consolidates the repeated definitional pattern. However, the expansion of composite components is an entirely mechanical process and could easily be automated.[1] For this reason, we claim that our omitted semantics for compositional components does not limit the applicability of our formalisms.

[1] In fact, this automated expansion of composite components is exactly what a simplifying pass of the code generator's parser already does. The resulting abstract syntax tree is only ever present in memory, but it would be a trivial modification to dump this to text as an equivalent, expanded specification.

```
/* System specification involving composite components */

procedure Processor {
  int process(in int input);
}

component Filter {
  provides Processor f;
}

component Pipe {
  provides Processor i;
  uses Processor o;
}

component Composite {
  provides Processor i;
  uses Processor o;

  provides Processor i2;
  uses Processor o2;

  composition {
    component Pipe p1;
    component Pipe p2;

    connection seL4RPCSimple conn(from p1.o, to p2.i);

    export p1.i -> i;
    export p2.o -> o;
  }
}

component Client {
  control;
  uses Processor inf;
}

assembly {
  composition {
    component Composite c1;
    component Composite c2;
    component Client client;
    component Filter filter;

    connection seL4RPCSimple conn1(from client.inf, to c1.i);
    connection seL4RPCSimple conn2(from c1.o, to c1.i2);
    connection seL4RPCSimple conn3(from c1.o2, to c2.i);
    connection seL4RPCSimple conn4(from c2.o, to c2.i2);
    connection seL4RPCSimple conn5(from c2.o2, to filter.f);
  }
}
```

```
/* System specification with composite components expanded */

procedure Processor {
  int process(in int input);
}

component Filter {
  provides Processor f;
}

component Pipe {
  provides Processor i;
  uses Processor o;
}

component Composite {
  provides Processor i2;
  uses Processor o2;
}

component Client {
  control;
  uses Processor inf;
}

assembly {
  composition {
    component Composite c1;
    component Composite c2;
    component Client client;
    component Filter filter;

    /* expanded from `Composite`'s definition */
    component Pipe c1_p1;
    component Pipe c1_p2;
    component Pipe c2_p1;
    component Pipe c2_p2;

    connection seL4RPCSimple conn1(from client.inf, to c1_p1.i);
    connection seL4RPCSimple conn2(from c1_p2.o, to c1.i2);
    connection seL4RPCSimple conn3(from c1.o2, to c2_p1.i);
    connection seL4RPCSimple conn4(from c2_p2.o, to c2.i2);
    connection seL4RPCSimple conn5(from c2.o2, to filter.f);

    /* expanded from `Composite`'s definition */
    connection seL4RPCSimple conn6(from c1_p1.o, to c1_p2.i);
    connection seL4RPCSimple conn7(from c2_p1.o, to c2_p2.i);
  }
}
```

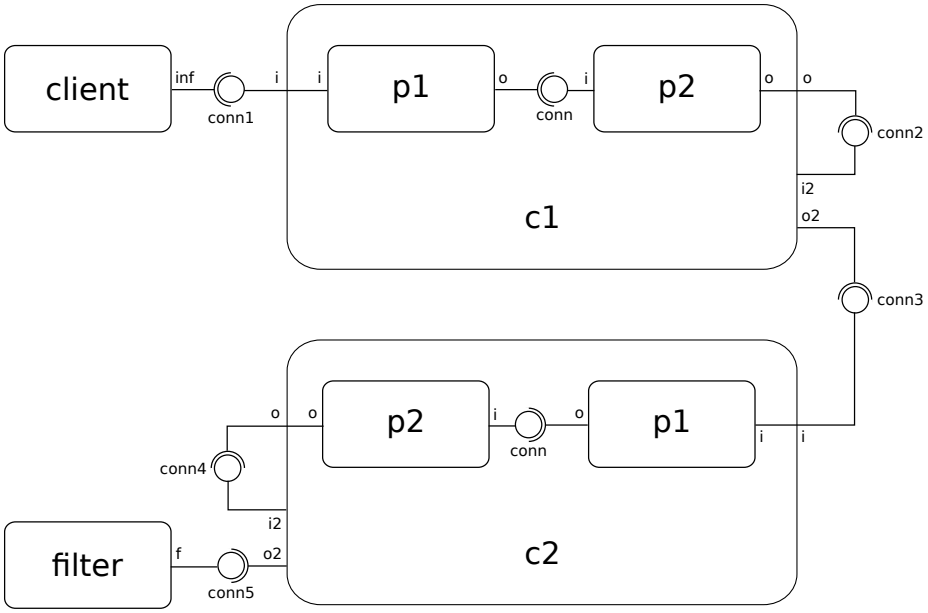

Figure A.1: An architecture containing composite components

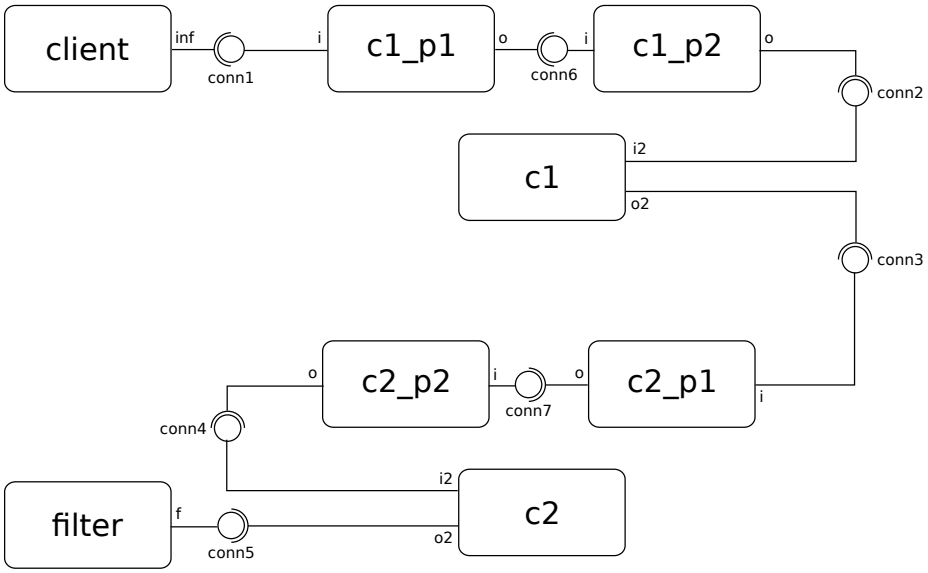

Figure A.2: An architecture with composite components expanded

Appendix B

Isabelle/HOL Locales

This appendix gives a brief introduction to an Isabelle/HOL feature called *locales*, by way of a simple example. The focus is on conveying the utility of locales as they are used in this work. For a more general and thorough treatment of locales, the interested reader is referred to the "Locales" section of the Isabelle/HOL reference manual (Wenzel 2014) and a dedicated tutorial (Ballarin 2010).

A locale allows you to define a context that carries a set of fixed, typed parameters and assumptions. It is introduced with the keyword, `locale`, and the fixed parameters and assumptions are given in the locale header.

```
1  locale foo =
2    fixes is_even :: "nat ⇒ bool"
3    assumes factor:"⋀x. is_even x ⟹ (∃y. 2 * y = x)"
4  begin
```

This locale fixes a single parameter, `is_even`, and assumes that this predicate implies that we can find a factor which, when doubled, equals the predicate argument. Inside the locale, we can prove lemmas that implicitly carry the assumptions of the locale. The assumptions themselves are available for use as if they were proven lemmas.

```
1  lemma even_imp_dvd:"is_even x ⟹ 2 dvd x"
2    apply (drule factor)
3    apply clarsimp
4    done
```

Though not stated, this lemma has the additional assumption, `factor`. In this sense, locales can be seen as shorthand to avoid repeating a set of common assumptions in every proof.

Having defined a locale, it can be *interpreted* to export instances of the contained lemmas. The process of interpretation requires that we provide terms to fit the "holes" of the fixed parameters and prove the assumptions of the locale. We can define the following function as a definition of even numbers.

```
1  primrec
2    g :: "nat ⇒ bool"
3  where
4    "g 0 = True"
5  | "g (Suc n) = (¬ g n)"
```

Now we go on to interpret the locale by proving that this function indeed satisfies the assumption, `factor`.

```
interpretation foo g
  apply unfold_locales
  apply (subst (asm) g_even_translate)
  apply (erule even_ex)
  done
```

We now have the lemma of the locale available for use, as instantiated with our g function.

```
lemma even_imp_dvd:"g x ⟹ 2 dvd x"
```

A key point to note is that the interpretation of the locale required a proof of `factor`, but not of `even_imp_dvd`. A user of the locale essentially gets the contained lemmas for the relatively low cost of discharging the locale assumptions.

Locales may be interpreted multiple times, independently. Thus they can be used for proof abstraction over common structures or types. For simplicity, our example has shown a locale containing a single fixed parameter, a single assumption and a single contained lemma, but it is possible to have an arbitrary number of all these three elements.

Appendix C

Multi-level Terminal Architecture Specification

What follows is the complete translated architectural specification of the multi-level terminal prototype from Chapter 8.

```
theory "MlsArch" imports
  "~~/../l4v/camkes/adl-spec/Types_CAMKES"
  "~~/../l4v/camkes/adl-spec/Library_CAMKES"
  "~~/../l4v/camkes/adl-spec/Wellformed_CAMKES"
begin

(* Procedure interfaces *)

definition
  Data :: procedure
where
  "Data ≡ [⦇ m_return_type = None,
             m_name = ''send_key'',
             m_parameters = [⦇ p_type = Primitive (Textual char),
                               p_direction = InParameter,
                               p_name = ''c'' ⦈] ⦈]"

lemma wf_Data: "wellformed_procedure Data"
  by code_simp

definition
  Paint :: procedure
where
  "Paint ≡ [⦇ m_return_type = Some (Primitive (Numerical Integer)),
              m_name = ''put_char'',
              m_parameters = [⦇ p_type = Primitive (Textual char),
                                p_direction = InParameter,
                                p_name = ''c'' ⦈] ⦈,
            ⦇ m_return_type = None,
              m_name = ''clear'',
              m_parameters = [] ⦈,
            ⦇ m_return_type = Some (Primitive (Numerical Integer)),
              m_name = ''move'',
              m_parameters = [⦇ p_type = Primitive (Numerical UnsignedInteger),
```

```
35                                  p_direction = InParameter,
36                                  p_name = ''x'' ⦆,
37                                ⦇ p_type = Primitive (Numerical UnsignedInteger),
38                                  p_direction = InParameter,
39                                  p_name = ''y'' ⦆] ⦆]"
40
41  lemma wf_Paint: "wellformed_procedure Paint"
42    by code_simp
43
44  (* Event interfaces *)
45
46  definition
47    Keypress :: event
48  where
49    "Keypress ≡ 0"
50
51  lemma wf_Keypress: "wellformed_event Keypress"
52    by code_simp
53
54  (* Dataport interfaces *)
55
56  definition
57    Input :: component
58  where
59    "Input ≡ ⦇ control = True,
60               requires = [(''low_output'', Data),
61                           (''high_output'', Data)],
62               provides = [],
63               dataports = [],
64               emits = [],
65               consumes = [],
66               attributes = [] ⦆"
67
68  lemma wf_Input: "wellformed_component Input"
69    by code_simp
70
71  definition
72    Low :: component
73  where
74    "Low ≡ ⦇ control = False,
75              requires = [(''low_display'', Paint)],
76              provides = [(''low_input'', Data)],
77              dataports = [],
78              emits = [(''key_out'', Keypress)],
79              consumes = [],
80              attributes = [] ⦆"
81
82  lemma wf_Low: "wellformed_component Low"
83    by code_simp
84
85  definition
86    High :: component
87  where
88    "High ≡ ⦇ control = True,
89               requires = [(''high_display'', Paint)],
90               provides = [(''high_input'', Data)],
91               dataports = [],
92               emits = [],
```

```
93              consumes = [(''key_in'', Keypress)],
94              attributes = [] ⦆"
95
96  lemma wf_High: "wellformed_component High"
97    by code_simp
98
99  definition
100   GUI :: component
101 where
102   "GUI ≡ ⦇ control = True,
103            requires = [],
104            provides = [(''low_data'', Paint),
105                        (''high_data'', Paint)],
106            dataports = [],
107            emits = [],
108            consumes = [],
109            attributes = [] ⦆"
110
111 lemma wf_GUI: "wellformed_component GUI"
112   by code_simp
113
114 definition
115   input :: component
116 where
117   "input ≡ Input"
118
119 lemma wf_input: "wellformed_component input"
120   by code_simp
121
122 definition
123   low :: component
124 where
125   "low ≡ Low"
126
127 lemma wf_low: "wellformed_component low"
128   by code_simp
129
130 definition
131   high :: component
132 where
133   "high ≡ High"
134
135 lemma wf_high: "wellformed_component high"
136   by code_simp
137
138 definition
139   gui :: component
140 where
141   "gui ≡ GUI"
142
143 lemma wf_gui: "wellformed_component gui"
144   by code_simp
145
146 definition
147   a :: connection
148 where
149   "a ≡ ⦇ conn_type = seL4RPCSimple,
150          conn_from = (''input'', ''low_output''),
```

```
151           conn_to = (''low'', ''low_input'') ⦆"
152
153   lemma wf_a: "wellformed_connection a"
154     by code_simp
155
156   definition
157     b :: connection
158   where
159     "b ≡ ⦇ conn_type = seL4RPCSimple,
160           conn_from = (''input'', ''high_output''),
161           conn_to = (''high'', ''high_input'') ⦆"
162
163   lemma wf_b: "wellformed_connection b"
164     by code_simp
165
166   definition
167     d :: connection
168   where
169     "d ≡ ⦇ conn_type = seL4AsynchNative,
170           conn_from = (''low'', ''key_out''),
171           conn_to = (''high'', ''key_in'') ⦆"
172
173   lemma wf_d: "wellformed_connection d"
174     by code_simp
175
176   definition
177     e :: connection
178   where
179     "e ≡ ⦇ conn_type = seL4RPCSimple,
180           conn_from = (''low'', ''low_display''),
181           conn_to = (''gui'', ''low_data'') ⦆"
182
183   lemma wf_e: "wellformed_connection e"
184     by code_simp
185
186   definition
187     f :: connection
188   where
189     "f ≡ ⦇ conn_type = seL4RPCSimple,
190           conn_from = (''high'', ''high_display''),
191           conn_to = (''gui'', ''high_data'') ⦆"
192
193   lemma wf_f: "wellformed_connection f"
194     by code_simp
195
196   definition
197     composition' :: composition
198   where
199     "composition' ≡ ⦇
200       components = [(''input'', input), (''low'', low), (''high'', high),
201         (''gui'', gui)],
202       connections = [(''a'', a), (''b'', b), (''d'', d), (''e'', e), (''f'', f)] ⦆"
203
204   lemma wf_composition': "wellformed_composition composition'"
205     by code_simp
206
207   definition
208     configuration' :: "configuration option"
```

```
209  where
210    "configuration' ≡ Some [(''input'', ''_domain'', ''1''),
211                            (''low'', ''_domain'', ''2''),
212                            (''low'', ''low_input_domain'', ''2''),
213                            (''high'', ''_domain'', ''3''),
214                            (''high'', ''high_input_domain'', ''3''),
215                            (''high'', ''key_in_domain'', ''3''),
216                            (''gui'', ''_domain'', ''4''),
217                            (''gui'', ''low_data_domain'', ''4''),
218                            (''gui'', ''high_data_domain'', ''4'')]"
219
220  lemma wf_configuration': "wellformed_configuration (the configuration')"
221    by code_simp
222
223  definition
224    assembly' :: assembly
225  where
226    "assembly' ≡ (| composition = composition',
227                    configuration = configuration' |)"
228
229  lemma wf_assembly': "wellformed_assembly assembly'"
230    by code_simp
231
232  end
```

Appendix D

Multi-level Terminal Behavioural Specification

Included below is the full generated behavioural specification of the multi-level terminal prototype from Chapter 8.

```
theory MlsCimp imports
  "~~/../l4v/camkes/glue-spec/Types"
  "~~/../l4v/camkes/glue-spec/Abbreviations"
  "~~/../l4v/camkes/glue-spec/Connector"
begin

datatype channel
  = a
  | b
  | d
  | e
  | f

datatype inst
  = input
  | low
  | high
  | gui
  | "d_e"

(* Input's interfaces *)
datatype Input_channel
  = Input_low_output
  | Input_high_output

definition
  Call_Input_low_output_send_key :: "(Input_channel ⇒ channel) ⇒
    ('cs local_state ⇒ char) ⇒ (channel, 'cs) comp"
where
  "Call_Input_low_output_send_key ch0' c_p ≡
     Request (λs1'. {⦇ q_channel = ch0' Input_low_output,
                       q_data = Call 0 [Char (c_p s1')] ⦈}) discard ;;
     Response (λs2' s3'. case q_data s2' of
         Return s4' ⇒ {(s3', ⦇ a_channel = ch0' Input_low_output,
```

```
                                a_data = Void |)}
       | _ ⇒ {})"

definition
  Call_Input_high_output_send_key :: "(Input_channel ⇒ channel) ⇒
    ('cs local_state ⇒ char) ⇒ (channel, 'cs) comp"
where
  "Call_Input_high_output_send_key ch5' cP ≡
     Request (λs6'. {(| q_channel = ch5' Input_high_output,
                        q_data = Call 0 [Char (cP s6')] |)}) discard ;;
     Response (λs7' s8'. case q_data s7' of
         Return s9' ⇒ {(s8', (| a_channel = ch5' Input_high_output,
                                a_data = Void |))}
       | _ ⇒ {})"

(* Low's interfaces *)
datatype Low_channel
  = Low_low_display
  | Low_low_input
  | Low_key_out

definition
  Call_Low_low_display_put_char :: "(Low_channel ⇒ channel) ⇒
    ('cs local_state ⇒ char) ⇒ ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Call_Low_low_display_put_char ch10' cP embed_ret ≡
     Request (λs11'. {(| q_channel = ch10' Low_low_display,
                         q_data = Call 0 [Char (cP s11')] |)}) discard ;;
     Response (λs12' s13'. case q_data s12' of
         Return s14' ⇒ {(embed_ret s13' (case hd s14' of Integer s15' ⇒ s15'),
                          (| a_channel = ch10' Low_low_display,
                            a_data = Void |))}
       | _ ⇒ {})"

definition
  Call_Low_low_display_clear :: "(Low_channel ⇒ channel) ⇒ (channel, 'cs) comp"
where
  "Call_Low_low_display_clear ch16' ≡
     Request (λs17'. {(| q_channel = ch16' Low_low_display,
                         q_data = Call 1 [] |)}) discard ;;
     Response (λs18' s19'. case q_data s18' of
         Return s20' ⇒ {(s19', (| a_channel = ch16' Low_low_display,
                                  a_data = Void |))}
       | _ ⇒ {})"

definition
  Call_Low_low_display_move :: "(Low_channel ⇒ channel) ⇒
    ('cs local_state ⇒ nat) ⇒ ('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Call_Low_low_display_move ch21' xP yP embed_ret ≡
     Request (λs22'. {(| q_channel = ch21' Low_low_display,
                         q_data = Call 2 [Number (xP s22'),
                                          Number (yP s22')] |)}) discard ;;
     Response (λs23' s24'. case q_data s23' of
         Return s25' ⇒ {(embed_ret s24' (case hd s25' of Integer s26' ⇒ s26'),
                          (| a_channel = ch21' Low_low_display,
```

```
                          a_data = Void |)}
      | _ ⇒ {})"

definition
  Recv_Low_low_input :: "(Low_channel ⇒ channel) ⇒
    ('cs local_state ⇒ char ⇒ 'cs local_state) ⇒ (channel, 'cs) comp ⇒
    (channel, 'cs) comp"
where
  "Recv_Low_low_input ch27' send_key_E Low_low_input_send_key ≡
     Response (λs28' s29'. case q_data s28' of
         Call s30' s31' ⇒
           (if s30' = 0
              then {(send_key_E s29' (case s31' ! 0 of Char s32' ⇒ s32'),
                     (| a_channel = ch27' Low_low_input,
                       a_data = Void |)}
              else {})
       | _ ⇒ {}) ;;
     Low_low_input_send_key ;;
     Request (λs33'. {(| q_channel = ch27' Low_low_input,
                        q_data = Return [] |)}) discard"

definition
  Emit_Low_key_out :: "(Low_channel ⇒ channel) ⇒ (channel, 'cs) comp"
where
  "Emit_Low_key_out ch34' ≡ EventEmit (ch34' Low_key_out)"

(* High's interfaces *)
datatype High_channel
  = High_high_display
  | High_high_input
  | High_key_in

definition
  Call_High_high_display_put_char :: "(High_channel ⇒ channel) ⇒
    ('cs local_state ⇒ char) ⇒ ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Call_High_high_display_put_char ch35' c_P embed_ret ≡
     Request (λs36'. {(| q_channel = ch35' High_high_display,
                        q_data = Call 0 [Char (c_P s36')] |)}) discard ;;
     Response (λs37' s38'. case q_data s37' of
         Return s39' ⇒ {(embed_ret s38' (case hd s39' of Integer s40' ⇒ s40'),
                         (| a_channel = ch35' High_high_display,
                           a_data = Void |)}
       | _ ⇒ {})"

definition
  Call_High_high_display_clear :: "(High_channel ⇒ channel) ⇒
    (channel, 'cs) comp"
where
  "Call_High_high_display_clear ch41' ≡
     Request (λs42'. {(| q_channel = ch41' High_high_display,
                        q_data = Call 1 [] |)}) discard ;;
     Response (λs43' s44'. case q_data s43' of
         Return s45' ⇒ {(s44', (| a_channel = ch41' High_high_display,
                                  a_data = Void |)}
       | _ ⇒ {})"

```

```
definition
  Call_High_high_display_move :: "(High_channel ⇒ channel) ⇒
    ('cs local_state ⇒ nat) ⇒ ('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Call_High_high_display_move ch46' xₚ yₚ embed_ret ≡
     Request (λs47'. {⦇ q_channel = ch46' High_high_display,
                        q_data = Call 2 [Number (xₚ s47'),
                                         Number (yₚ s47')] ⦈}) discard ;;
     Response (λs48' s49'. case q_data s48' of
         Return s50' ⇒ {(embed_ret s49' (case hd s50' of Integer s51' ⇒ s51'),
                         ⦇ a_channel = ch46' High_high_display,
                           a_data = Void ⦈)}
       | _ ⇒ {})"

definition
  Recv_High_high_input :: "(High_channel ⇒ channel) ⇒
    ('cs local_state ⇒ char ⇒ 'cs local_state) ⇒ (channel, 'cs) comp ⇒
    (channel, 'cs) comp"
where
  "Recv_High_high_input ch52' send_keyₑ High_high_input_send_key ≡
     Response (λs53' s54'. case q_data s53' of
         Call s55' s56' ⇒
           (if s55' = 0
              then {(send_keyₑ s54' (case s56' ! 0 of Char s57' ⇒ s57'),
                     ⦇ a_channel = ch52' High_high_input,
                       a_data = Void ⦈)}
              else {})
       | _ ⇒ {}) ;;
     High_high_input_send_key ;;
     Request (λs58'. {⦇ q_channel = ch52' High_high_input,
                        q_data = Return [] ⦈}) discard"

definition
  Poll_High_key_in :: "(High_channel ⇒ channel) ⇒ ('cs local_state ⇒ bool ⇒
    'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Poll_High_key_in ch59' embed60' ≡ EventPoll (ch59' High_key_in) embed60'"

definition
  Wait_High_key_in :: "(High_channel ⇒ channel) ⇒ ('cs local_state ⇒ bool ⇒
    'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Wait_High_key_in ch59' embed61' ≡ EventWait (ch59' High_key_in) embed61'"

(* GUI's interfaces *)
datatype GUI_channel
  = GUI_low_data
  | GUI_high_data

definition
  Recv_GUI_low_data :: "(GUI_channel ⇒ channel) ⇒
    ('cs local_state ⇒ char ⇒ 'cs local_state) ⇒ (channel, 'cs) comp ⇒
    ('cs local_state ⇒ int) ⇒ (channel, 'cs) comp ⇒
    ('cs local_state ⇒ nat ⇒ nat ⇒ 'cs local_state) ⇒ (channel, 'cs) comp ⇒
    ('cs local_state ⇒ int) ⇒ (channel, 'cs) comp"
where
  "Recv_GUI_low_data ch62' put_charₑ GUI_low_data_put_char
```

```
209       put_char_returnP GUI_low_data_clear moveE GUI_low_data_move move_returnP ≡
210        (Response (λs63' s64'. case q_data s63' of
211             Call s65' s66' ⇒
212               (if s65' = 0
213                  then {(put_charE s64' (case s66' ! 0 of Char s67' ⇒ s67'),
214                    ⦇ a_channel = ch62' GUI_low_data,
215                      a_data = Void ⦈)}
216                  else {})
217           | _ ⇒ {}) ;;
218         GUI_low_data_put_char ;;
219         Request (λs68'. {⦇ q_channel = ch62' GUI_low_data,
220                           q_data = Return [Integer (put_char_returnP s68')] ⦈})
221                   discard)
222        ⊔
223        (Response (λs69' s70'. case q_data s69' of
224             Call s71' s72' ⇒
225               (if s71' = 1
226                  then {(s70', ⦇ a_channel = ch62' GUI_low_data,
227                                 a_data = Void ⦈)}
228                  else {})
229           | _ ⇒ {}) ;;
230         GUI_low_data_clear ;;
231         Request (λs73'. {⦇ q_channel = ch62' GUI_low_data,
232                           q_data = Return [] ⦈}) discard)
233        ⊔
234        (Response (λs74' s75'. case q_data s74' of
235             Call s76' s77' ⇒
236               (if s76' = 2
237                  then {(moveE s75' (case s77' ! 0 of Number s78' ⇒ s78')
238                          (case s77' ! 1 of Number s79' ⇒ s79'),
239                         ⦇ a_channel = ch62' GUI_low_data,
240                           a_data = Void ⦈)}
241                  else {})
242           | _ ⇒ {}) ;;
243         GUI_low_data_move ;;
244         Request (λs80'. {⦇ q_channel = ch62' GUI_low_data,
245                           q_data = Return [Integer (move_returnP s80')] ⦈})
246                   discard)"
247
248 definition
249   Recv_GUI_high_data :: "(GUI_channel ⇒ channel) ⇒
250     ('cs local_state ⇒ char ⇒ 'cs local_state) ⇒ (channel, 'cs) comp ⇒
251     ('cs local_state ⇒ int) ⇒ (channel, 'cs) comp ⇒
252     ('cs local_state ⇒ nat ⇒ nat ⇒ 'cs local_state) ⇒ (channel, 'cs) comp ⇒
253     ('cs local_state ⇒ int) ⇒ (channel, 'cs) comp"
254 where
255   "Recv_GUI_high_data ch81' put_charE GUI_high_data_put_char put_char_returnP
256      GUI_high_data_clear moveE GUI_high_data_move move_returnP ≡
257       (Response (λs82' s83'. case q_data s82' of
258            Call s84' s85' ⇒
259              (if s84' = 0
260                 then {(put_charE s83' (case s85' ! 0 of Char s86' ⇒ s86'),
261                        ⦇ a_channel = ch81' GUI_high_data,
262                          a_data = Void ⦈)}
263                 else {})
264           | _ ⇒ {}) ;;
265         GUI_high_data_put_char ;;
266         Request (λs87'. {⦇ q_channel = ch81' GUI_high_data,
```

```
                        q_data = Return [Integer (put_char_returnP s87')] |)})
                discard)
      ⊔
      (Response (λs88' s89'. case q_data s88' of
           Call s90' s91' ⇒
             (if s90' = 1
                then {(s89', (| a_channel = ch81' GUI_high_data,
                                a_data = Void |))}
                else {})
         | _ ⇒ {}) ;;
       GUI_high_data_clear ;;
       Request (λs92'. {(| q_channel = ch81' GUI_high_data,
                        q_data = Return [] |)}) discard)
      ⊔
      (Response (λs93' s94'. case q_data s93' of
           Call s95' s96' ⇒
             (if s95' = 2
                then {(moveE s94' (case s96' ! 0 of Number s97' ⇒ s97')
                        (case s96' ! 1 of Number s98' ⇒ s98'),
                       (| a_channel = ch81' GUI_high_data,
                        a_data = Void |))}
                else {})
         | _ ⇒ {}) ;;
       GUI_high_data_move ;;
       Request (λs99'. {(| q_channel = ch81' GUI_high_data,
                        q_data = Return [Integer (move_returnP s99')] |)})
                discard)"

(* Component instantiations *)

definition
  Call_input_low_output_send_key :: "('cs local_state ⇒ char) ⇒
    (channel, 'cs) comp"
where
  "Call_input_low_output_send_key ≡
     Call_Input_low_output_send_key (λs100'. case s100' of
         Input_low_output ⇒ a
       | Input_high_output ⇒ b)"

definition
  Call_input_high_output_send_key :: "('cs local_state ⇒ char) ⇒
    (channel, 'cs) comp"
where
  "Call_input_high_output_send_key ≡
     Call_Input_high_output_send_key (λs101'. case s101' of
         Input_low_output ⇒ a
       | Input_high_output ⇒ b)"

definition
  Call_low_low_display_put_char :: "('cs local_state ⇒ char) ⇒
    ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Call_low_low_display_put_char ≡
     Call_Low_low_display_put_char (λs102'. case s102' of
         Low_low_input ⇒ a
       | Low_key_out ⇒ d
       | Low_low_display ⇒ e)"
```

```
definition
  Call_low_low_display_clear :: "(channel, 'cs) comp"
where
  "Call_low_low_display_clear ≡
     Call_Low_low_display_clear (λs103'. case s103' of
         Low_low_input ⇒ a
       | Low_key_out ⇒ d
       | Low_low_display ⇒ e)"

definition
  Call_low_low_display_move :: "('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ nat) ⇒ ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Call_low_low_display_move ≡
     Call_Low_low_display_move (λs104'. case s104' of
         Low_low_input ⇒ a
       | Low_key_out ⇒ d
       | Low_low_display ⇒ e)"

definition
  Recv_low_low_input :: "('cs local_state ⇒ char ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp ⇒ (channel, 'cs) comp"
where
  "Recv_low_low_input ≡
     Recv_Low_low_input (λs105'. case s105' of
         Low_low_input ⇒ a
       | Low_key_out ⇒ d
       | Low_low_display ⇒ e)"

definition
  Emit_low_key_out :: "(channel, 'cs) comp"
where
  "Emit_low_key_out ≡
     Emit_Low_key_out (λs106'. case s106' of
         Low_low_input ⇒ a
       | Low_key_out ⇒ d
       | Low_low_display ⇒ e)"

definition
  Call_high_high_display_put_char :: "('cs local_state ⇒ char) ⇒
    ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Call_high_high_display_put_char ≡
     Call_High_high_display_put_char (λs107'. case s107' of
         High_high_input ⇒ b
       | High_key_in ⇒ d
       | High_high_display ⇒ f)"

definition
  Call_high_high_display_clear :: "(channel, 'cs) comp"
where
  "Call_high_high_display_clear ≡
     Call_High_high_display_clear (λs108'. case s108' of
         High_high_input ⇒ b
       | High_key_in ⇒ d
       | High_high_display ⇒ f)"

```

```
383 definition
384   Call_high_high_display_move :: "('cs local_state ⇒ nat) ⇒
385     ('cs local_state ⇒ nat) ⇒ ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
386     (channel, 'cs) comp"
387 where
388   "Call_high_high_display_move ≡
389      Call_High_high_display_move (λs109'. case s109' of
390          High_high_input ⇒ b
391        | High_key_in ⇒ d
392        | High_high_display ⇒ f)"
393
394 definition
395   Recv_high_high_input :: "('cs local_state ⇒ char ⇒ 'cs local_state) ⇒
396     (channel, 'cs) comp ⇒ (channel, 'cs) comp"
397 where
398   "Recv_high_high_input ≡
399      Recv_High_high_input (λs110'. case s110' of
400          High_high_input ⇒ b
401        | High_key_in ⇒ d
402        | High_high_display ⇒ f)"
403
404 definition
405   Poll_high_key_in :: "('cs local_state ⇒ bool ⇒ 'cs local_state) ⇒
406     (channel, 'cs) comp"
407 where
408   "Poll_high_key_in ≡
409      Poll_High_key_in (λs111'. case s111' of
410          High_high_input ⇒ b
411        | High_key_in ⇒ d
412        | High_high_display ⇒ f)"
413
414 definition
415   Wait_high_key_in :: "('cs local_state ⇒ bool ⇒ 'cs local_state) ⇒
416     (channel, 'cs) comp"
417 where
418   "Wait_high_key_in ≡
419      Wait_High_key_in (λs112'. case s112' of
420          High_high_input ⇒ b
421        | High_key_in ⇒ d
422        | High_high_display ⇒ f)"
423
424 definition
425   Recv_gui_low_data :: "('cs local_state ⇒ char ⇒ 'cs local_state) ⇒
426     (channel, 'cs) comp ⇒ ('cs local_state ⇒ int) ⇒ (channel, 'cs) comp ⇒
427     ('cs local_state ⇒ nat ⇒ nat ⇒ 'cs local_state) ⇒ (channel, 'cs) comp ⇒
428     ('cs local_state ⇒ int) ⇒ (channel, 'cs) comp"
429 where
430   "Recv_gui_low_data ≡
431      Recv_GUI_low_data (λs113'. case s113' of
432          GUI_low_data ⇒ e
433        | GUI_high_data ⇒ f)"
434
435 definition
436   Recv_gui_high_data :: "('cs local_state ⇒ char ⇒ 'cs local_state) ⇒
437     (channel, 'cs) comp ⇒ ('cs local_state ⇒ int) ⇒ (channel, 'cs) comp ⇒
438     ('cs local_state ⇒ nat ⇒ nat ⇒ 'cs local_state) ⇒ (channel, 'cs) comp ⇒
439     ('cs local_state ⇒ int) ⇒ (channel, 'cs) comp"
440 where
```

```
441     "Recv_gui_high_data ≡
442        Recv_GUI_high_data (λs114'. case s114' of
443            GUI_low_data ⇒ e
444          | GUI_high_data ⇒ f)"
445
446   locale system_state =
447     fixes init_component_state :: 'cs
448     fixes trusted :: "(inst, ((channel, 'cs) comp × 'cs local_state)) map"
449   begin
450
451   definition
452     Input_untrusted :: "(Input_channel ⇒ channel) ⇒ (channel, 'cs) comp"
453   where
454     "Input_untrusted ch115' ≡
455        LOOP (
456            UserStep
457          ⊔ ArbitraryRequest (ch115' Input_low_output)
458          ⊔ ArbitraryResponse (ch115' Input_low_output)
459          ⊔ ArbitraryRequest (ch115' Input_high_output)
460          ⊔ ArbitraryResponse (ch115' Input_high_output))"
461
462   definition
463     Low_untrusted :: "(Low_channel ⇒ channel) ⇒ (channel, 'cs) comp"
464   where
465     "Low_untrusted ch116' ≡
466        LOOP (
467            UserStep
468          ⊔ ArbitraryRequest (ch116' Low_low_display)
469          ⊔ ArbitraryResponse (ch116' Low_low_display)
470          ⊔ ArbitraryRequest (ch116' Low_low_input)
471          ⊔ ArbitraryResponse (ch116' Low_low_input)
472          ⊔ ArbitraryRequest (ch116' Low_key_out)
473          ⊔ ArbitraryResponse (ch116' Low_key_out))"
474
475   definition
476     High_untrusted :: "(High_channel ⇒ channel) ⇒ (channel, 'cs) comp"
477   where
478     "High_untrusted ch117' ≡
479        LOOP (
480            UserStep
481          ⊔ ArbitraryRequest (ch117' High_high_display)
482          ⊔ ArbitraryResponse (ch117' High_high_display)
483          ⊔ ArbitraryRequest (ch117' High_high_input)
484          ⊔ ArbitraryResponse (ch117' High_high_input)
485          ⊔ ArbitraryRequest (ch117' High_key_in)
486          ⊔ ArbitraryResponse (ch117' High_key_in))"
487
488   definition
489     GUI_untrusted :: "(GUI_channel ⇒ channel) ⇒ (channel, 'cs) comp"
490   where
491     "GUI_untrusted ch118' ≡
492        LOOP (
493            UserStep
494          ⊔ ArbitraryRequest (ch118' GUI_low_data)
495          ⊔ ArbitraryResponse (ch118' GUI_low_data)
496          ⊔ ArbitraryRequest (ch118' GUI_high_data)
497          ⊔ ArbitraryResponse (ch118' GUI_high_data))"
498
```

```
(* Simulated component for event Keypress. *)
type_synonym Keypress_channel = unit

definition
  Keypress :: "(Keypress_channel ⇒ channel) ⇒ (channel, 'cs) comp"
where
  "Keypress ch119' ≡ event (ch119' ())"

(* Component instantiations *)

definition
  input_untrusted :: "(channel, 'cs) comp"
where
  "input_untrusted ≡
     Input_untrusted (λs120'. case s120' of
         Input_low_output ⇒ a
       | Input_high_output ⇒ b)"

definition
  low_untrusted :: "(channel, 'cs) comp"
where
  "low_untrusted ≡
     Low_untrusted (λs121'. case s121' of
         Low_low_input ⇒ a
       | Low_key_out ⇒ d
       | Low_low_display ⇒ e)"

definition
  high_untrusted :: "(channel, 'cs) comp"
where
  "high_untrusted ≡
     High_untrusted (λs122'. case s122' of
         High_high_input ⇒ b
       | High_key_in ⇒ d
       | High_high_display ⇒ f)"

definition
  gui_untrusted :: "(channel, 'cs) comp"
where
  "gui_untrusted ≡
     GUI_untrusted (λs123'. case s123' of
         GUI_low_data ⇒ e
       | GUI_high_data ⇒ f)"

definition
  "d_e_instance" :: "(channel, 'cs) comp"
where
  "d_e_instance ≡ Keypress (λ_. d)"

(* Global initial state *)
definition
  "gs₀" :: "(inst, channel, 'cs) global_state"
where
  "gs₀ p124' ≡ case trusted p124' of Some s125' ⇒
      Some s125'
    | _ ⇒ (case p124' of
          input ⇒ Some (input_untrusted, Component init_component_state)
        | low ⇒ Some (low_untrusted, Component init_component_state)
```

```
557         | high ⇒ Some (high_untrusted, Component init_component_state)
558         | gui ⇒ Some (gui_untrusted, Component init_component_state)
559         | d_e ⇒ Some (d_e_instance, init_event_state))"
560 
561 end
562 
563 end
```