CAmkES Glue Code Semantics

Matthew Fernandez, Peter Gammie, June Andronick, Gerwin Klein, Ihor Kuz

April 2013

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. NICTA is also funded and supported by the Australian Capital Territory, the New South Wales, Queensland and Victorian Governments, the Australian National University, the University of New South Wales, the University of Melbourne, the University of Queensland, the University of Sydney, Griffith University, Queensland University of Technology, Monash University and other university partners.

This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S.Government.

Abstract

This document describes the formal dynamic semantics of CAmkES glue code, in particular of the communication stubs generated for components at compile time. The semantics is based on a simple concurrent imperative language with message passing that is easy to extend and instantiate for specific applications. Instead of one generic semantics for all systems, we take the approach of generating a high-level semantic description for each specific ADL component specification to ease verification of specific systems in the future.

We show the definitions and types for expressing components and glue code, and provide some examples of generated Isabelle theories with synchronous, asynchronous, and shared memory communication.

Contents

1 Introduction

This document shows the formal Isabelle/HOL [3] specification of the behaviour of CAmkES component systems defined by ADL descriptions [2]. This formal specification is intended to apply to the glue code; the generated communication stubs that are provided to the user by the CAmkES platform. It extends and supplements the previous report which described the static semantics of component and system specifications [1]. Together these two reports form the formal specification of the CAmkES ADL. Although this is its most interesting part, the specification presented here goes beyond providing just the glue code semantics. Instead we provide an abstract high-level specification of the behaviour of an entire CAmkES component system.

The specification is high-level, because it abstracts from the underlying kernel mechanisms and message formats. Instead, it is based on a general concurrent message passing framework that can transmit messages of arbitrary high-level types. Instantiating this framework we restrict it to the kinds of message types of the ADL description and map CAmkES mechanisms to the message passing primitives. Showing that the kernel and glue code indeed implement this high-level semantic view is the main proof obligation of the future glue code correctness proof.

The idea of the specification is to provide a high-level view of the behaviour of a component system using semantic mechanisms that nevertheless map reasonably easily to the glue code implementation and the underlying kernel mechanisms that provide architecture and communication boundary enforcement. The basic communication principle of the underlying semantic framework is synchronous message passing. This is presented in a way that makes it convenient to additionally model atomic asynchronous events and shared memory reads/writes by adding intermediate simulated components. These intermediate model processes map to kernel event buffers and the usual behaviour of shared memory pages. At the expense of making the shared memory component more complex, it would be feasible to explicitly include the effects of weak memory models. We do not do this here, because the intended application scenario is a unicore setting.

Similar to how the instantiation of a component system is generated from its ADL description in conjunction with provided user code, we generate the formal specification of a complete CAmkES component system from the same ADL description together with a set of generic base definitions in this document and a set of user-provided behaviour definitions for trusted components. Trusted components are those that are claimed to be more constrained in their behaviour than the architecture boundaries enforce.

The remainder of this report is structured as follows. We first introduce the semantic concurrency framework the glue code definitions are based on, in Chapter 2. We then proceed to define the basic data types that instantiate this semantic framework to CAmkES systems, in Chapter 3. After this, in Chapter 4 we can define the building blocks that the generated glue code specifications will further instantiate and use. Chapter 5 defines the intermediate event and memory components mentioned above and Chapter 6 provides default instantiations of types and definitions that the user may choose to override.

Chapter 7, Chapter 8 and Chapter 9 show example specifications produced from a number of small CAmkES ADL descriptions, illustrating the output of the generation phase. These examples show the actual glue code specifications that fit the corresponding generated C code. A more detailed system is presented in Chapter 10 to illustrate how to define and use trusted components.

2 Concurrent Imperative Syntax and Semantics

This chapter introduces a small concurrent imperative language with synchronous message passing. The sequential part of the language is a standard, minimal, Turing-complete While-language. It is sufficient to express the semantics of CAmkES glue code and the behaviour of small trusted components. It can be extended easily with procedures and other programming language concepts in standard ways if the behaviour of larger trusted components needs to be described. For this document, we concentrate on the ADL and glue code semantics and keep the language as simple as possible.

The message passing mechanism is a slight variation of standard synchronous message passing instructions `send` and `receive` that would map directly to seL4 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 slightly to the instructions `Request` and `Receive` 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, e.g. 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 for instance, where the response from memory is instantaneous, or to model asynchronous messages, where the effect is simply to store the message in a buffer.

Below follows the datatype 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"`

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

The alternatives of the data type are the usual:

- Skip, which does nothing,
- a local operation that can model any function on the local state `'s`, such as a variable assignment for instance,
- standard sequential composition,
- standard if-then-else,
- standard while loops with a boolean expression and a body,
- binary non-deterministic choice,

- message sending (request),
- and finally message receiving (response).

In Isabelle, this is:

```
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 notational 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.

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.

```
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 judgment 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 top: 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 nondeterministic imperative language. Local and terminating steps produce τ transitions, all other labels are passed through appropriately.

```
inductive small_step ::
  "('a, 'q, 's) com × 's ⇒ ('a, 'q) seq_label ⇒
   ('a, 'q, 's) com × 's ⇒ bool"
  ("_ →_ _" [55, 0, 56] 55)
where
  Request:
  "⟦ α ∈ action s; s' ∈ val β s ⟧ ⟹
  (Request action val, s) →≪α, β≫ (SKIP, s')"
| Response:
  "(s', β) ∈ action α s ⟹ (Response action, s) →≫α, β≪ (SKIP, s')"

| LocalOp:
  "s' ∈ R s ⟹ (LocalOp R, s) →τ (SKIP, s')"

| Seq1:
  "(c1, s) →α (SKIP, s') ⟹ (c1;; c2, s) →α (c2, s')"
| Seq2:
  "⟦ (c1, s) →α (c1', s'); c1' ≠ SKIP ⟧ ⟹ (c1;; c2, s) →α (c1';; c2, s')"

| IfTrue:
  "⟦ b s; (c1, s) →α (c1', s') ⟧ ⟹ (IF b THEN c1 ELSE c2, s) →α (c1', s')"
| IfFalse:
  "⟦ ¬ b s; (c2, s) →α (c2', s') ⟧ ⟹ (IF b THEN c1 ELSE c2, s) →α (c2', s')"

| WhileTrue:
  "⟦ b s; (c, s) →α (c', s') ⟧ ⟹
  (WHILE b DO c, s) →α (c';; WHILE b DO c, s')"
| WhileFalse:
  "¬ b s ⟹ (WHILE b DO c, s) →τ (SKIP, s)"

| Choose1:
  "(c1, s) →α (c1', s') ⟹ (c1 ⊔ c2, s) →α (c1', s')"
| Choose2:
  "(c2, s) →α (c2', s') ⟹ (c1 ⊔ c2, s) →α (c2', 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 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 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 as either any enabled internal τ step of any process, or as a communication step between two processes. For such 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.

```
inductive
  system_step ::
  "('a, 'proc, 'q, 's) global_state ⇒ ('a, 'proc, 'q, 's) global_state ⇒ bool"
  ("_ → _" [55, 56] 55)
where
  LocalStep:
  "⟦ gs p →τ c'; gs' = gs(p := c') ⟧ ⟹ gs → gs'"
| CommunicationStep:
  "⟦gs p₁ →≪α, β≫ c₁'; gs p₂ →≫α, β≪ c₂'; p₁ ≠ p₂;
    gs' = gs(p₁ := c₁', p₂ := c₂') ⟧
  ⟹ gs → gs'"
```

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

This concludes the definition of the small concurrent imperative base language. In the following, we use this language to express the high-level semantics of CAmkES ADL glue code as it maps to the seL4 microkernel.

3 Datatypes

This chapter builds up the basic data types that are necessary to cast CAmkES glue code in terms of the concurrent imperative language. In particular, we define data types for the kinds of variables 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.

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 the ADL model, with a value attached.

```
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.

Event and shared memory connections are modelled using an intermediate component. This is explained in more detail in Chapter 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 implementation.

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 `unit`.

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 indicies) 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"
```

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"
```

3.4 Global State

The global state of a system is a mapping from component instance identifiers to a pair of component (i.e. 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"
```

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.

4.1 Local Component Operations

4.1.1 UNIV_c

The set of all possible states a component can be in. This is essentially any local state with the exception of the states representing events and shared memory.

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

4.1.2 Internal Step

An internal step in a component that arbitrarily modifies its own local state. Note that it is not possible for an event or shared memory component to take this step.

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

4.1.3 User Steps

A representation of `internal` in the concurrent imperative language. That is, an arbitrary local step.

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

4.2 Communication Component Operations

4.2.1 Discard Messages

Receive a `Void` message and do nothing in reaction.

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 {}"
```

4.2.2 Arbitrary Requests

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

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

4.2.3 Arbitrary Responses

Non-deterministically receive any message on a given channel. The command receives any message, makes a nondeterministic local state update, and returns the set of all possible responses β on the given channel.

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

4.2.4 Event Emit

Emit an event. The command sends the message `Set` on the given channel and discards any response to model asynchronous behaviour with respect to the event buffer components. The message is independent of the local state `s`.

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

4.2.5 Event Poll

Poll for a pending event from an asynchronous buffer 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`. This payload is embedded in the local state of the component using the user-provided function `embed`.

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

4.2.6 Event Wait

Wait for a pending event. The command takes a channel `c`, and embedding function `embed` (see above). 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 ≡
    Request (λ_. {⦇q_channel = c, q_data = Poll⦈})
            (λa s. case a_data a of Pending b ⇒ if b then {embed s b} else {}
                                  | _ ⇒ {})"
```

4.2.7 Memory Read

Read from a shared memory location. 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 of 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 ≡
    Request (λs. {⦇q_channel = c, q_data = Read (addr s)⦈})
            (λa s. case a_data a of Value v ⇒ {embed s v} | _ ⇒ {})"
```

4.2.8 Memory Write

Write 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"
```

This concludes the list of the basic operations from which the glue code is composed. We now proceed to define the intermediate communication components for events and shared memory.

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.

5.1 Event Components

We represent a CAmkES event connector as a component always listening for `Set` or `Poll` questions that then simultaneously 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`.

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.

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

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-place 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 (the (s addr))|))}
     | 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.

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

In CAmkES ADL 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 $\texttt{Buf}_d$`_channel = unit`

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

6 Component Behaviour

The definitions of a full system are expected to come from a combination of generated and user-provided theories. The CAmkES generator utility creates a base theory using the types and definitions previously discussed that defines primitive operations of a specific system. The user is then expected to provide a theory that defines the trusted components of the system, building on top of these definitions. The generator also produces a theory describing the system as a whole that builds on top of the user's intermediate theory. Final reasoning about system properties is expected to be done in another theory building on this generated system theory.

These theory dependencies are depicted in Figure 6.1.

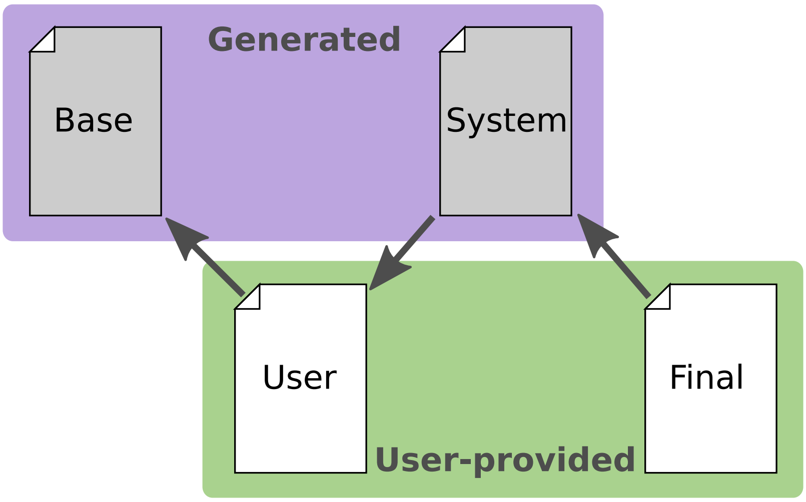

Figure 6.1: Theorem dependencies

The remainder of this section describes the default contents of the intermediate user theory if none other is provided.

When using the generated theories, the user is expected to provide the following type instantiations and definitions:

- A type for `component_state` representing 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.

If parts of this are unnecessary for the user's aims, then they can import the default implementations described below.

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 the type `unit` captures the irrelevance of this.

```
type_synonym component_state = unit
```

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.

```
definition
  init_component_state :: component_state
where
  "init_component_state ≡ ()"
```

6.2 Untrusted Components

Any component which does not have its definition supplied will be given a generated definition that allows it to non-deterministically perform any local operation 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.

6.3 Trusted Components

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. The types `component` and `lstate` are overridden in the generated theories and do not need to be provided here or by the user, but they make the definition of `trusted` more readable.

```
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"
```

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 components defined by the following CAmkES specification:

```
procedure Simple {
  smallstring echo_string(in smallstring s);
  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 seL4RPC simple(from client.s, to echo.s);
  }
}
```

The system can be depicted as two components connected with a single interface, in Figure 7.1.

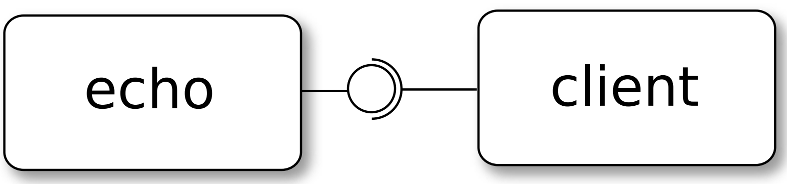

Figure 7.1: Hello world

The types and definitions presented in this section are semantically identical to those generated for the system above. However, the order in which entities are introduced and the white space has been modified for better readability.

7.1 Generated Base Theory

7.1.1 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`.

7.1.2 Interface Primitives

This section describes 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 three methods, `echo_string`, `echo_int` and `echo_parameter`, hence three 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_string` takes a `smallstring` input parameter, `s`, which necessitates a projection function, $\mathtt{s}_P$, as a parameter to `Call_Client_s_echo_string`. Conversely, the method returns a `smallstring` parameter, necessitating an embedding function, `embed` 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_string :: "(Client_channel ⇒ channel) ⇒
    ('cs local_state ⇒ string) ⇒
    ('cs local_state ⇒ string ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Call_Client_s_echo_string ch s_P embed ≡
    Request (λs. {⦇q_channel = ch Client_s,
      q_data = Call 0 (String (s_P s) # [])⦈}) discard ;;
    Response (λq s. case q_data q of Return xs ⇒
      {(embed s (case hd xs of String v ⇒ v),
      ⦇a_channel = ch Client_s, a_data = Void⦈)} | _ ⇒ {})"

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 i_P embed ≡
    Request (λs. {⦇q_channel = ch Client_s,
      q_data = Call 1 (Integer (i_P s) # [])⦈}) discard ;;
    Response (λq s. case q_data q of Return xs ⇒
      {(embed 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 pin_P embed ≡
    Request (λs. {⦇q_channel = ch Client_s,
      q_data = Call 2 (Integer (pin_P s) # [])⦈}) discard ;;
    Response (λq s. case q_data q of Return xs ⇒
      {(embed 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 defintion 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_string parameter is expected to be the user's implementation of the echo_string method.

```
definition
  Recv_Echo_s :: "(Echo_channel ⇒ channel) ⇒
    ('cs local_state ⇒ string ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp ⇒ ('cs local_state ⇒ string) ⇒
    ('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_string_E Echo_s_echo_string echo_string_return_P
     echo_int_E Echo_s_echo_int echo_int_return_P echo_parameter_E
     Echo_s_echo_parameter echo_parameter_return_P echo_parameter_pout_P ≡
    (Response (λq s. case q_data q of Call n xs ⇒
      (if n = 0 then {(echo_string_E s (case xs ! 0 of String v ⇒ v),
        (|a_channel = ch Echo_s, a_data = Void|))} else {}) | _ ⇒ {}) ;;
     Echo_s_echo_string ;;
     Request (λs. {(|q_channel = ch Echo_s,
       q_data = Return (String (echo_string_return_P s) # [])|)}) discard)
    ⊔
    (Response (λq s. case q_data q of Call n xs ⇒
      (if n = 1 then {(echo_int_E 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_return_P s) # [])|)}) discard)
    ⊔
    (Response (λq s. case q_data q of Call n xs ⇒
      (if n = 2 then {(echo_parameter_E s (case xs ! 0 of Integer v ⇒ v),
        (|a_channel = ch Echo_s, a_data = Void|))} else {}) | _ ⇒ {}) ;;
     Echo_s_echo_parameter ;;
     Request (λs. {(|q_channel = ch Echo_s,
       q_data = Return (Integer (echo_parameter_return_P s) #
                        Integer (echo_parameter_pout_P s) # [])|)}) discard)"
```

7.1.3 Instantiations of Primitives

With the component type definitions in place, the definitions of component instance primitives are much simpler as they are partial applications of the component type definitions. A call definition is generated for each method in each outgoing interface in each component instance that partially applies the call definitions described in Section 7.1.2 with a mapping from the component's interface to the system connection.

The parameter used to specialise the component 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 λ_. `simple`, but for simplicity the generator does not make this optimisation.

```
definition
  Call_client_s_echo_string :: "('cs local_state ⇒ string) ⇒
    ('cs local_state ⇒ string ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp"
where
  "Call_client_s_echo_string ≡
    Call_Client_s_echo_string (λc. case c of Client_s ⇒ simple)"

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 ⇒ string ⇒ 'cs local_state) ⇒
    (channel, 'cs) comp ⇒ ('cs local_state ⇒ string) ⇒
    ('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)"
```

7.2 Generated System Theory

7.2.1 Types

At the system level, type instantiations are provided for components and local and global state. These are derived by simply instantiating the relevant types with the types generated in the base theory. Note that the `component_state` type is expected to be provided by the user in their intermediate theory.

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

type_synonym lstate = "component_state local_state"

type_synonym gstate = "(inst, channel, component_state) global_state"
```

7.2.2 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, in that they permit an untrusted component to perform actions such as sending on an incoming interface which may not be possible in the implementation.

Recall from Chapter 6 that the user is expected to provide a mapping describing trusted components in their intermediate theory. A definition of untrusted execution for each component is generated regardless of whether all instances of that component in the current system have trusted specifications or not.

```
definition
  Client_untrusted :: "(Client_channel ⇒ channel) ⇒ component"
where
  "Client_untrusted ch ≡
    LOOP (
      UserStep
    ⊔ ArbitraryRequest (ch Client_s)
    ⊔ ArbitraryResponse (ch Client_s))"

definition
  Echo_untrusted :: "(Echo_channel ⇒ channel) ⇒ component"
where
  "Echo_untrusted ch ≡
    LOOP (
      UserStep
    ⊔ ArbitraryRequest (ch Echo_s)
    ⊔ ArbitraryResponse (ch Echo_s))"
```

7.2.3 Component Instances

As was the case for the instantiation of primitives in Section 7.1.3, 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 :: component
where
  "client_untrusted ≡ Client_untrusted (λc. case c of Client_s ⇒ simple)"
```

```
definition
  echo_untrusted :: component
where
  "echo_untrusted ≡ Echo_untrusted (λc. case c of Echo_s ⇒ simple)"
```

7.2.4 Initial State

The final generated definition is the initial state of the system. Following the type instantiations in Section 7.2.1, 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 (user-provided) mapping of trusted instances and, if it does not find this, falls back on the generated untrusted definitions.

```
definition
  gs0 :: gstate
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))"
```

8 Example – Events

This section provides an example following on from Chapter 7 that gives an example of the corresponding 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 seL4Asynch simpleEvent1(from source.ev, to sink.ev);
  }
}
```

8.1 Generated Base Theory

8.1.1 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
  | simpleEvent1$_e$

datatype Collector_channel

```
  = Collector_ev
```

datatype `Emitter_channel`
```
  = Emitter_ev
```

8.1.2 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` functions must be supplied by the user to save the result of the operation into the component's local state.

Event callbacks are not currently represented. These can be represented by hand in the intermediate user theory. We plan to extend the generator in future to wrap this functionality in a primitive for the user.

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

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

For each component type with an `emits` interface, a single primitive is generated to correpond 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)"
```

8.1.3 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)"
```

8.2 Generated System Theory

8.2.1 Types

Identical types to those presented in Section 7.2.1 are generated for a system involving events.

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

type_synonym `lstate = "component_state local_state"`

type_synonym `gstate = "(inst, channel, component_state) global_state"`

8.2.2 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.

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))"
```

8.2.3 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`, described in Section 5.1.

```
type_synonym SomethingHappenedEvent_channel = unit

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

8.2.4 Component Instances

The definitions of untrusted component instances are generated as in Chapter 7, 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
  simpleEvent1_e_instance :: component
where
  "simpleEvent1_e_instance ≡ SomethingHappenedEvent (λ_. simpleEvent1)"
```

8.2.5 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 $\texttt{simpleEvent1}_e$ 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))"
```

9 Example – Dataports

This section provides an example of generated types and definitions derived from a CAmkES dataport interface. The following example system is used throughout this section:

```
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 also, unlike the previous two examples, contains a component type that is instantiated twice.

9.1 Generated Base Theory

9.1.1 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.

datatype `channel`
  = `simple2`
  | `simple1`

datatype `inst`
  = `comp2`
  | `comp1`
  | `simple2`$_d$
  | `simple1`$_d$

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

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

9.1.2 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`.

Read and write are unconditionally generated for each dataport interface because all dataports are read/write memory regions. Should the CAmkES model be extended to allow read-only or write-only dataports only the relevant single operation would be generated here.

```
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 ≡
    MemoryRead (ch DataportTest_d2) addr embed"

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"

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 ≡
   MemoryRead (ch DataportTest_d1) addr embed"

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"
```

9.1.3 Instantiations of Primitives

The specialisation of the primitives from Section 9.1.2 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_comp2_d2 :: "('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ variable ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Read_comp2_d2 ≡
    Read_DataportTest_d2 (λc. case c of DataportTest_d1 ⇒ simple2
                                  | DataportTest_d2 ⇒ simple1)"

definition
  Write_comp2_d2 :: "('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ variable) ⇒ (channel, 'cs) comp"
where
  "Write_comp2_d2 ≡
    Write_DataportTest_d2 (λc. case c of DataportTest_d1 ⇒ simple2
                                   | DataportTest_d2 ⇒ simple1)"

definition
  Read_comp2_d1 :: "('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ variable ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Read_comp2_d1 ≡
    Read_DataportTest_d1 (λc. case c of DataportTest_d1 ⇒ simple2
                                  | DataportTest_d2 ⇒ simple1)"

definition
  Write_comp2_d1 :: "('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ variable) ⇒ (channel, 'cs) comp"
where
  "Write_comp2_d1 ≡
    Write_DataportTest_d1 (λc. case c of DataportTest_d1 ⇒ simple2
                                   | DataportTest_d2 ⇒ simple1)"

definition
  Read_comp1_d2 :: "('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ variable ⇒ 'cs local_state) ⇒ (channel, 'cs) comp"
where
  "Read_comp1_d2 ≡
    Read_DataportTest_d2 (λc. case c of DataportTest_d2 ⇒ simple2
                                  | DataportTest_d1 ⇒ simple1)"

definition
  Write_comp1_d2 :: "('cs local_state ⇒ nat) ⇒
    ('cs local_state ⇒ variable) ⇒ (channel, 'cs) comp"
```

where
```
"Write_comp1_d2 ≡
  Write_DataportTest_d2 (λc. case c of DataportTest_d2 ⇒ simple2
                                     | DataportTest_d1 ⇒ simple1)"
```

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
```
"Write_comp1_d1 ≡
  Write_DataportTest_d1 (λc. case c of DataportTest_d2 ⇒ simple2
                                     | DataportTest_d1 ⇒ simple1)"
```

9.2 Generated System Theory

9.2.1 Types

At the system level we have the now familiar generated types.

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

type_synonym `lstate = "component_state local_state"`

type_synonym `gstate = "(inst, channel, component_state) global_state"`

9.2.2 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))"
```

9.2.3 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)"
```

9.2.4 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
```
simple2_d_instance :: component
```
where
```
"simple2_d_instance ≡ Buf_d (λ_. simple2)"
```

definition
```
simple1_d_instance :: component
```
where
```
"simple1_d_instance ≡ Buf_d (λ_. simple1)"
```

9.2.5 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.

definition
```
gs_0 :: gstate
```
where
```
"gs_0 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))"
```

10 Example – System Level Reasoning

This section provides an example of a more detailed CAmkES system and reasoning about a system level property of such a system. The example system is described by the following specification:

```
procedure Lookup {
  smallstring get_value(in smallstring id);
};

component Client {
  control;
  uses Lookup l;
}

component Store {
  provides Lookup l;
}

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

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

    connection seL4RPC one(from client.l, to filter.external);
    connection seL4RPC two(from filter.backing, to store.l);
  }
}
```

It consists of an instance, `client`, that reads values out of a key-value store in the instance, `store`. Its access is mediated by the instance `filter` that prevents it reading the value “baz” associated with the key “secret”.

The generated types and definitions are omitted, but they are similar to those described in Chapter 7.

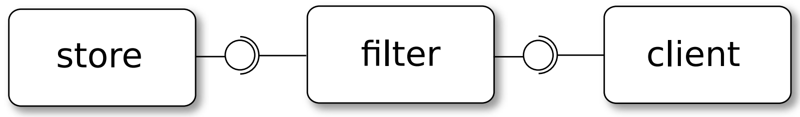

Figure 10.1: Example filter system

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.

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.

```
lemma "∀c.
  ¬(communicates_on c client_untrusted ∧ communicates_on c store_untrusted)"
  unfolding communicates_on_def client_untrusted_def Client_untrusted_def
            store_untrusted_def Store_untrusted_def
  apply clarsimp
  unfolding UserStep_def ArbitraryRequest_def ArbitraryResponse_def
  apply clarsimp
  apply (case_tac c, clarsimp+)
  done
```

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.

10.2 Behavioural Properties

To reason about the behaviour of components themselves, we need to provide more information in the intermediate user theory. In this section we present an example of this and a proof that `client` never receives the secret, “baz”. 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 reponses as only the value “bar” or the empty string, “”.

```
definition
  filter_responses :: "channel question set"
where
  "filter_responses ≡ {x. ∃v ∈ {''bar'', ''''}. q_data x = Return [String 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)]"
```

Now it's possible to state and prove the desired property of the system; that `client` never receives the secret "baz".

```
lemma "∀p. ∃e s. gs₀ p = Some (e, s) ∧
  (e = client_untrusted ∨
  ¬(∃c. sends e {x. q_channel x = c ∧ q_data x = Return [String ''baz'']} ∧
  receives_on c client_untrusted))"
  unfolding gs₀_def trusted_def apply clarsimp
  apply (case_tac p, clarsimp)
    unfolding store_untrusted_def Store_untrusted_def apply clarsimp
    unfolding UserStep_def ArbitraryRequest_def ArbitraryResponse_def
    apply clarsimp
    unfolding client_untrusted_def Client_untrusted_def apply clarsimp
    unfolding UserStep_def ArbitraryRequest_def ArbitraryResponse_def
    apply clarsimp
   apply clarsimp
  apply clarsimp
  unfolding filter_trusted_def UserStep_def ArbitraryRequest_def
            ArbitraryResponse_def apply clarsimp
  unfolding filter_responses_def apply clarsimp
  done
```

Bibliography

[1] Ihor Kuz, Matthew Fernandez, Gerwin Klein, and Toby Murray. CAmkES manual and formalisation. Technical report, NICTA, October 2012.

[2] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. 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):687–699, May 2007.

[3] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.