CAmkES Formalisation of a Component Platform

Matthew Fernandez, Gerwin Klein, Ihor Kuz, Toby Murray

October 2012

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.

Contents

1 Introduction

CAmkES is a component platform for embedded microkernel-based systems, offering many of the standard features available in component platforms. Some relevant features of CAmkES that are not common to all component platforms are:

- *Explicit composite components.* CAmkES components can be assembled to form a re-usable composite, that can then be referenced within a containing system.
- *Multiple instantiation of a single component.* Multiple copies of a component can exist within a system, distinguished by different identifiers.
- *Multiple implementations of an interface.* A single component can implement an interface more than once. This can be useful for providing a dedicated interface for each client or functionally different implementations.
- *Distinction between active and passive components.* Components can have a thread of control or be invoked via an event loop. CAmkES distinguishes these modes of operation at an architectural level.
- *"Provides" interfaces can be left unsatisfied at runtime.* When a component implements a given interface, it does not have to be connected to a component that uses that interface. The converse is not true.
- *Hardware devices are components.* Hardware devices can be specified in ADL that generates interface methods for accessing the device.

2 Types

This section describes the types of entities that appear in the CAmkES Interface Definition Language (IDL) and Architecture Description Language (ADL). The model assumes that all syntactic elements are available in a single environment; that is, all `import` inclusion statements have already been processed and all references are to be resolved within this context.

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 method names and parameters. These symbols are used during code generation and therefore need to be distinct within a specific interface.

type_synonym `idl_symbol = string`

The second type of symbols are 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. E.g. Instantiated interfaces within separate components can have an identical name.

type_synonym `adl_symbol = string`

Although both symbols map to the same underlying type, these have different constraints (e.g. IDL symbols become direct substrings of code-level symbols and hence need to respect symbol naming restrictions for the target language(s)) and will be extended to represent these in a future iteration of this specification.

2.2 Methods

Methods are the elements that make up a CAmkES trait (described below). Each method within a CAmkES trait has a list of parameters defined by a type, direction and symbol name. Each method also has an optional return value type. The valid types for method parameters and return values include a set of high level types designed to abstract the types available in a general programming language. By using only these types in a trait description, the interface can be implemented in any valid target language.

When fixed width types are required for an interface there are a set of types available that are C-specific. Using these types in a trait description precludes implementing or using the trait in a component not written in C. In general the high-level types should be used in preference to the C-specific types.

Array types are supported as method parameters and return types in CAmkES in two flavours: arrays with a given size and arrays terminated by a `NULL` (`0`) value. Both types of arrays are parameterised with the underlying type of their elements. 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 C-specific type prevents implementing or using it in a component not written in C. Arrays of arrays and multidimensional arrays are not supported.

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
```

Note that a string in C is a `NULL`-terminated character array. If a C-specific string is required in a trait it is best to specify it manually (i.e. as a `TerminatedArray (Textual char)`).

```
datatype primitive =
    Numerical number
  | Textual textual
```

```
datatype array =
    SizedArray primitive
  | TerminatedArray primitive
```

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

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

```
datatype param_direction =
    InParameter
  | OutParameter
  | InOutParameter
```

Each trait comprises a collection of methods that each have an optional return type, identifier and a list of parameters. Each parameter has a type and an identifier.

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

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

The translation from trait methods in IDL to their representation in Isabelle is straightforward. The CAmkES method

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

is translated to the Isabelle representation

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

More examples are given in Chapter 4.

2.3 Interfaces

Connections between two components are made from one interface to another. CAmkES distinguishes between interfaces that consist of a list of function calls and interfaces that have other patterns of interaction.

There are three basic types of supported interfaces. The first, `trait`, is used for modelling traditional caller-callee semantics of interaction. The second, `event` is used for asynchronous notifications such as interrupts. Finally, `dataport` is used to model shared memory communication.

```
type_synonym procedure = "method list"
type_synonym event     = nat — ID
type_synonym dataport  = "string option" — type
datatype interface =
    Procedure procedure
  | Event event
  | Dataport dataport
```

2.4 Connectors

Two components are connected via a connector. The type of a connector is an abstraction of the underlying communication mechanism. Connectors come in three distinct types, native connectors, hardware connectors and export connectors.

Native connectors map directly to implementation mechanisms. These are the types of connectors that are found in almost all component platform models. The event-style connectors, `AsynchronousEvent` and `RPCEvent` are used to model communication consisting of an identifier with no associated message data.

```
datatype native_connector_type =
    AsynchronousEvent — an asynchronous notification
```

| `RPCEvent` — a synchronous notification
| `RPC` — a synchronous channel
| `DirectCall` — a synchronous channel to a component in the same address space

| `SharedData` — a shared memory region

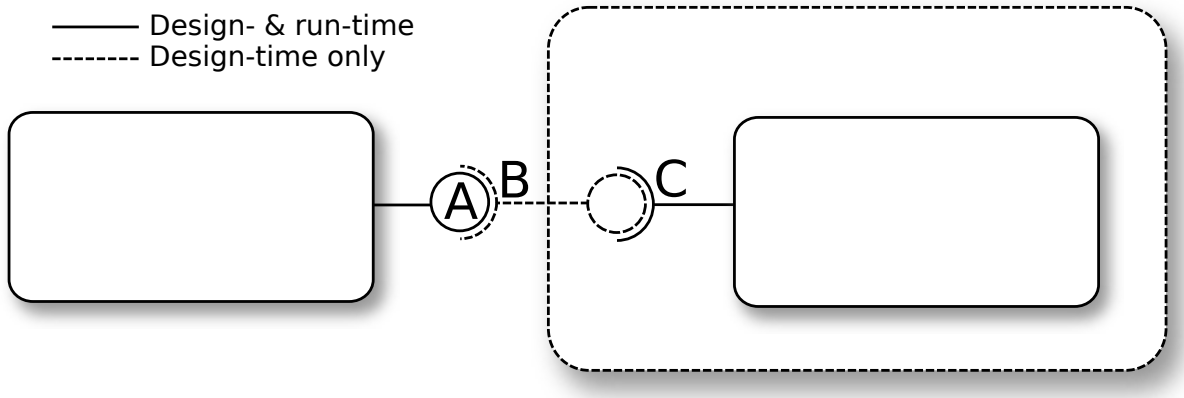
Recalling that hardware devices are modelled as components in CAmkES, hardware connectors are used to connect the interface of a device to the interface of a software component. Devices must be connected using the connector type that corresponds to the mode of interaction with the device.

datatype `hardware_connector_type =`
`HardwareMMIO` — memory mapped IO
| `HardwareInterrupt` — device interrupts
| `HardwareIOPort` — IA32 IO ports

Export connectors are used when specifying a compound component. A compound component has a set of interfaces that are a subset of the unconnected interfaces of its constituent components. The exposed interfaces of the compound component are defined by using export connectors to map these to the interfaces of the internal components.

Export connectors are purely an architectural-level entity and do not exist at runtime. During code generation connections through exported interfaces are flattened. That is, a connection from interface A to exported interface B that exports interface C is instantiated as a connection from interface A to interface C would be. They can be thought of as a design-level convenience.

Figure 2.1: An export connector



datatype `export_connector_type =`
`ExportAsynchronous`
| `ExportRPC`
| `ExportData`

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

Connectors are distinguished by the mode of interaction they enable. The reason for this will become clearer in Section 3.3.

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

2.5 Components

Functional entities in a CAmkES system are represented as components. These are re-usable collections of source code with explicit descriptions of the exposed methods of interaction (`interfaces` described above).

Components have three distinct modes of communication:

1. Synchronous communication over procedures. This communication is analogous to a function call and happens over a channel established from a `requires` interface to a `provides` interface.
2. Asynchronous communication using events. This is suitable for things like interrupts 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`.

```
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"
```

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)"
```

A composition block is used to contain all components 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, as indicated above. These attributes are accessible to the component at the code level at runtime.

```
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 re-use of compositions and configurations.

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

2.7 Future Work

2.7.1 Component Hierarchy

Some component platforms support the notion of explicit composite components. This allows a composition of components to be re-used as a component itself. In the context of the model presented above, this would allow a `composition` element to appear anywhere a `component` element is expected. Flexibility like this is desirable to avoid repeatedly constructing common design patterns involving fine grained components. There are plans to extend CAmkES to add this functionality.

2.7.2 Interface Arrays

When specifying a more complicated dynamic component, it can be desirable to define an array of interfaces. For example, a component that provides an arbitrary number of copies of a specified procedure. This would be implemented by the size of the array (the number of such copies) being made available to the component at runtime and an index being provided with each procedure method invocation. An example of how this could be useful is discussed in Section 4.4.

Extending this further, allowing the specification of interface arrays that can be resized at runtime, by adding or removing connections, enables even greater flexibility. Supporting this kind of dynamism in a system requires meta functions (for modifying the interface array) and introduces further complexity in handling failures of these.

3 Wellformedness of Specifications

To prove that a system specification is correct, we need to define what correctness means for the entities that can exist in a CAmkES specification. This section provides a definition of wellformedness for each syntactic element that captures the necessary conditions for it to represent a valid system configuration. Any wellformed system is capable of being instantiated in a way that corresponds to its ADL.

3.1 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.

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.

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

The implementation currently only supports 32 distinct events (0 - 31). This limitation may be removed in a future iteration.

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

Dataports do not have any attributes beyond their type, so their wellformedness is trivial.

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

definition
```
  wellformed_interface :: "interface ⇒ bool"
```
where
```
  "wellformed_interface i ≡ (case i of
```

```
    Procedure p ⇒ wellformed_procedure p
  | Event e ⇒ wellformed_event e
  | Dataport d ⇒ wellformed_dataport d)"
```

3.2 Components

For a component to be valid internally, its interfaces must not conflict and must themselves be wellformed.

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)))"
```

3.3 Connectors

For a connector to be valid its mode of interaction must be consistent with the underlying mechanism.

definition

```
  wellformed_connector :: "connector ⇒ bool"
```

where

```
  "wellformed_connector c ≡ (case c of
    SyncConnector t ⇒ (case t of
      Native n ⇒ n ∈ {RPCEvent, RPC, DirectCall}
     |Hardware h ⇒ h ∈ {HardwareIOPort}
     |Export e ⇒ e ∈ {ExportRPC})
   |AsyncConnector t ⇒ (case t of
      Native n ⇒ n ∈ {AsynchronousEvent}
     |Hardware h ⇒ h ∈ {HardwareInterrupt}
     |Export e ⇒ e ∈ {ExportAsynchronous})
   |MemoryConnector t ⇒ (case t of
      Native n ⇒ n ∈ {SharedData}
     |Hardware h ⇒ h ∈ {HardwareMMIO}
     |Export e ⇒ e ∈ {ExportData}))"
```

3.4 Connections

definition

```
  wellformed_connection :: "connection ⇒ bool"
```

where

```
  "wellformed_connection c ≡ True"
```

3.5 ADL Symbol Resolution

All procedures must be satisfied by a unique connection. To define unique connections, we first define a general predicate `ex_one` that is true when a predicate is satisfied for precisely one element in a list.

definition
```
  ex_one :: "'a list ⇒ ('a ⇒ bool) ⇒ bool"
```
where
```
  "ex_one xs P ≡ length (filter P xs) = 1"
```

The following two declarations provide more convenience for this function. `∃1 x ∈ xs. P x` will be translated into `ex_one xs P`.

syntax
```
  "_ex_one" :: "pttrn ⇒ 'a set ⇒ bool ⇒ bool" ("(4∃1 _∈_./ _)" [0, 0, 10] 10)
```
translations
```
  "∃1 x∈xs. P" == "CONST ex_one xs (λx. P)"
```

We can now define valid procedures. For each procedure `x` there must be precisely one connection `y` that fits the component instance.

definition
```
  refs_valid_procedures ::
    "adl_symbol ⇒ (adl_symbol × procedure) list ⇒
     (adl_symbol × connection) list ⇒ bool"
```
where
```
  "refs_valid_procedures component_instance procedures conns ≡
    ∀x ∈ set procedures.
     (∃1 y ∈ conns. from_component (snd y) = component_instance ∧
                    from_interface (snd y) = fst x)"
```

For events and dataports, an interface can be left unconnected in a system with no adverse effects.

definition
```
  refs_valid_components ::
    "(adl_symbol × component) list ⇒ (adl_symbol × connection) list ⇒ bool"
```
where
```
  "refs_valid_components comps conns ≡
    ∀x ∈ set comps. refs_valid_procedures (fst x) (requires (snd x)) conns"
```

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"
```

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)"
```

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))"
```

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

definition
```
  wellformed_assembly :: "assembly ⇒ bool"
```
where
```
  "wellformed_assembly a ≡
```

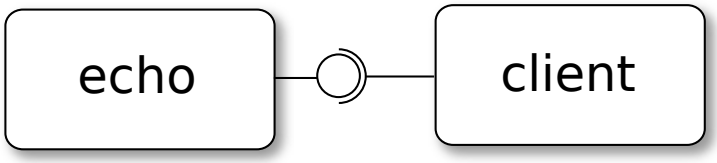
```
wellformed_composition (composition a) ∧ (case (configuration a) of
  None ⇒ True
| Some x ⇒ wellformed_configuration x)"
```

4 Example Systems

4.1 Echo

The following ADL and IDL describe an example system involving two components, client and echo. There is a single connection between them, from a procedure `client.s` that client requires to a procedure `echo.s` that echo provides. The system is depicted in Figure 4.1.

Figure 4.1: Echo example



The procedure used in this system is expressed by the following IDL:

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

The representation of this in Isabelle is quite similar:[1]

```
definition
  simple :: procedure
where
  "simple ≡ [
    ⦇ m_return_type = Some (Primitive (Textual String)), m_name = ''echo_string'',
      m_parameters = [
        ⦇ p_type = Primitive (Textual String),
          p_direction = InParameter,
          p_name = ''s'' ⦈
      ] ⦈,
    ⦇ m_return_type = Some (Primitive (Numerical Integer)), m_name = ''echo_int'',
      m_parameters = [
        ⦇ p_type = Primitive (Numerical Integer),
          p_direction = InParameter,
          p_name = ''i'' ⦈
      ] ⦈,
```

[1] The procedure parameter types `int` and `uint` are synonyms for `integer` and `unsigned integer`, respectively, and are therefore not modelled in Isabelle.

```
  ⦇ m_return_type = None, m_name = ''echo_parameter'', m_parameters = [
      ⦇ p_type = Primitive (Numerical Integer),
        p_direction = InParameter,
        p_name = ''pin'' ⦈,
      ⦇ p_type = Primitive (Numerical Integer),
        p_direction = InParameter,
        p_name = ''pout'' ⦈
    ] ⦈
]"
```

Each component of the system is described by a separate IDL representation:

```
component Client {
  control;
  uses Simple s;
}

component Echo {
  provides Simple s;
}
```

These generate the following formal representations in Isabelle:

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

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

A `composition` block is used to combine these elements into the complete system. There are no attributes in this simple system so the `configuration` block of the `assembly` can be omitted. The two components are connected via a seL4RPC connection. Note that the underlying implementation mechanism of this seL4RPC connection is abstracted.

```
assembly {
  composition {
    component Echo echo;
    component Client client;
    connection seL4RPC simple(from client.s, to echo.s);
  }
}
```

Once again the generated Isabelle formalism looks similar:

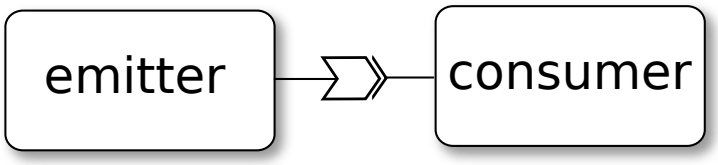
definition
```
  system :: assembly
```
where
```
  "system ≡ ⦇
    composition = ⦇
      components = [(''echo'', echo),(''client'', client)],
      connections = [(''simple'', ⦇
        conn_type = seL4RPC,
        conn_from = (''client'', ''s''),
        conn_to = (''echo'', ''s'')
      ⦈)]
    ⦈,
    configuration = None
  ⦈"
```

Since our wellformedness conditions are executable, we can now prove that this example is a wellformed assembly by evaluation.

lemma `"wellformed_assembly system"` **by** `eval`

4.2 Events

Figure 4.2: Event example



The following example shows a system using a single event to provide asynchronous communication between two components. The identifier assigned to the event, `1`, is unimportant in this example as there is only one event in use.

definition
```
  signal :: event
```
where
```
  "signal ≡ 1"
```

The active component `emitter` generates events of the type `signal`.

definition

```
  emitter :: component
where
  "emitter ≡ (|
    control = True,
    requires = [],
    provides = [],
    dataports = [],
    emits = [(''event'', signal)],
    consumes = [],
    attributes = []
  |)"
```

The component `consumer` expects to receive these events. When a component is defined to consume an event, a function for registering a callback for this event is made available to the component. The component is initialised at runtime with no callback registered to allow it to do any necessary setup without needing to guard against concurrency. Thus, even when consuming components are conceptually passive they are usually specified as active (`control = True`) with an entry function that performs some initialisation and then registers an event handler.

definition

```
  consumer :: component
where
  "consumer ≡ (|
    control = True,
    requires = [],
    provides = [],
    dataports = [],
    emits = [],
    consumes = [(''event'', signal)],
    attributes = []
  |)"
```

The system assembly looks similar to that shown in Section 4.1, but an asynchronous connector is used between the components.

definition

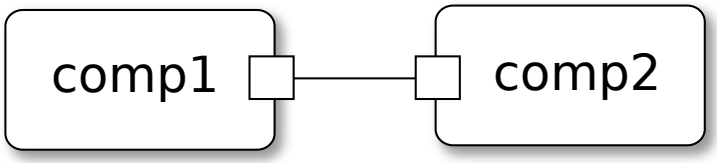
```
  event_system :: assembly
where
  "event_system ≡ (|
    composition = (|
      components = [(''source'', emitter), (''sink'', consumer)],
      connections = [(''simpleEvent1'', (|
        conn_type = seL4Asynch,
        conn_from = (''source'', ''event''),
        conn_to = (''sink'', ''event'')
      |))]
    |),
    configuration = None
  |)"
```

Again, wellformedness is proved easily by evaluation.

lemma `"wellformed_assembly event_system"` **by** `eval`

4.3 Dataport Usage

Figure 4.3: Dataport example



The following example demonstrates the use of a shared memory region, referred to as a dataport in CAmkES. It also uses one of the key aspects of a component platform, component re-use. First the definition of a simple component that uses two dataports:

definition

```
  data_client :: component
```

where

```
  "data_client ≡ ⦇
    control = True,
    requires = [],
    provides = [],
    dataports = [(''d1'', None), (''d2'', None)],
    emits = [],
    consumes = [],
    attributes = []
  ⦈"
```

By instantiating this component twice (once as `comp1` and once as `comp2`) the system contains two identical components. The assembly below connects the first dataport of `comp1` to the second dataport of `comp2` and vice versa. It is possible to specify a system that instantiates `data_client` once and connects one of the instance's dataports to the other, but there is nothing to be gained from a component communicating with itself via shared memory.

definition

```
  data_system :: assembly
```

where

```
  "data_system ≡ ⦇
    composition = ⦇
      components = [(''comp1'', data_client), (''comp2'', data_client)],
      connections = [(''simple1'', ⦇
        conn_type = seL4SharedData,
        conn_from = (''comp1'', ''d1''),
        conn_to = (''comp2'', ''d2'')
      ⦈), (''simple2'', ⦇
        conn_type = seL4SharedData,
        conn_from = (''comp2'', ''d1''),
        conn_to = (''comp1'', ''d2'')
      ⦈)]
    ⦈,
    configuration = None
  ⦈"
```

The data port example is wellformed:

lemma `"wellformed_assembly data_system"` **by** `eval`

4.4 Secure Terminal

This section presents a more realistic component system as a prototype of a secure terminal. Two components are each given a separate region of a text terminal to which they can write character data. They accomplish this by using a connection to a third, trusted component that manages the terminal.

Figure 4.4: Terminal example

The interface for writing to the terminal takes coordinates to write to and a single character to write. The coordinates are relative to the caller's dedicated region. That is, (0, 0) represents the upper left corner of the caller's region, not the terminal as a whole. The method `put_char` returns 0 on success and non-zero if the coordinates are out of range.

```
definition
  display :: procedure
where
  "display ≡ [
    ⦇ m_return_type = Some (Primitive (Numerical uint32_t)), m_name = ''put_char'',
      m_parameters = [
        ⦇ p_type = Primitive (Numerical uint32_t),
          p_direction = InParameter,
          p_name = ''x'' ⦈,
        ⦇ p_type = Primitive (Numerical uint32_t),
          p_direction = InParameter,
          p_name = ''y'' ⦈,
        ⦇ p_type = Primitive (Numerical uint32_t),
          p_direction = InParameter,
          p_name = ''data'' ⦈
      ] ⦈ ]"
```

The trusted component that manages the terminal is passive and executes only in response to `put_char` calls from its clients. The component described below supports exactly two components. This is a case where a more flexible definition would be possible using interface arrays as described in Section 2.7.2.

```
definition
  manager :: component
where
  "manager ≡ ⦇
    control = False,
    requires = [],
    provides = [(''domain1'', display), (''domain2'', display)],
    dataports = [],
    emits = [],
    consumes = [],
```

```
    attributes = []
  ⦆"
```

The definition of the client adds an attribute so the execution can branch based on which instance of the component is running, but the instances could equally well execute exactly the same code and have their (identical) output written to the two distinct regions by the manager.

definition
```
  terminal_client :: component
```
where
```
  "terminal_client ≡ ⦇
    control = True,
    requires = [(''d'', display)],
    provides = [],
    dataports = [],
    emits = [],
    consumes = [],
    attributes = [(''ID'', Primitive (Numerical Integer))]
  ⦈"
```

Each client is connected to a single interface of the manager.

definition
```
  channel1 :: connection
```
where
```
  "channel1 ≡ ⦇
    conn_type = seL4RPC,
    conn_from = (''client1'', ''d''),
    conn_to = (''manager'', ''domain1'')
  ⦈"
```
definition
```
  channel2 :: connection
```
where
```
  "channel2 ≡ ⦇
    conn_type = seL4RPC,
    conn_from = (''client2'', ''d''),
    conn_to = (''manager'', ''domain2'')
  ⦈"
```

definition
```
  comp :: composition
```
where
```
  "comp ≡ ⦇
    components = [(''manager'', manager),
                  (''client1'', terminal_client),
                  (''client2'', terminal_client)],
    connections = [(''channel1'', channel1),
                   (''channel2'', channel2)]
  ⦈"
```

Each client is given a unique identifier so it can distinguish itself. As noted above, this is not necessarily required in all systems with multiple instantiations of a single component.

definition
```
  conf :: configuration
```

where
```
  "conf ≡ [(''client1'', ''ID'', ''1''),
           (''client2'', ''ID'', ''2'')]"
```

definition
```
  terminal :: assembly
```
where
```
  "terminal ≡ ⦇
    composition = comp,
    configuration = Some conf
  ⦈"
```

Wellformedness for this more complex example is easy as well.

lemma `"wellformed_assembly terminal"` **by** `eval`