

## Part I

### FRAMING THE PROBLEM AND CURRENT STATE-OF-THE-ART

You can put some informational part preamble text here.  
Illo principalmente su nos. Non message *occidental* anglo-romanica da. Debitas effortio simplificate sia se, auxiliari summarios da que, se avantiate publicationes via. Pan in terra summarios, capital interlingua se que. Al via multo esser specimen, campo responder que da. Le usate medical addresses pro, europa origine sanctificate nos se.



## Part II

### DESIGN ITERATIONS AND CONSTRUCTING A THEORY

You can put some informational part preamble text here.  
Illo principalmente su nos. Non message *occidental* anglo-romanica da. Debitas effortio simplificate sia se, auxiliari summarios da que, se avantiate publicationes via. Pan in terra summarios, capital interlingua se que. Al via multo esser specimen, campo responder que da. Le usate medical addresses pro, europa origine sanctificate nos se.



## Part III

### CONTRIBUTIONS AND EVALUATION

In this part of the thesis, the more general concepts and techniques that can be applied to ubiquitous computing are described. These concepts and techniques were extracted from work done during the three design iterations.



*Whenever we capture the complexity of the real world in formal structures, whether language, social structures, or computer systems, we are creating discrete tokens for continuous and fluid phenomena. In doing so, we are bound to have difficulty. However, it is only in doing these things that we can come to understand, to have valid discourse, and to design.*

— Alan Dix

In order to share device capabilities with other devices in the environment, we require ways to describe these capabilities. While we have touched lightly on some of the techniques in the thesis so far, this chapter will focus in more detail on the current state-of-the-art, as well as how we extended these techniques to create a new way of modelling device capabilities using ontologies.

Most of the existing work on modelling interaction capabilities focuses on Graphical User Interface (GUI) based techniques. A *universal user interface language* describes user interfaces that are rendered by mapping a description's device-independent interaction elements to a target platform's concrete interface objects [62]. This allows developers to create the user interface in an abstract language without targeting a specific device. Examples of interface languages include User Interface Markup Language (UIML), Extensible Interface Markup Language (XIML), Carnegie Mellon University's Personal Universal Controller (PUC) and the International Committee for Information Technology Standards Universal Remote Console (INCITS/V2 URC). These languages allow devices to determine the most suitable presentation based on a predefined set of abstract user interface components.

UIML maps interface elements to target UI objects using a styling section, resulting in one styling section per target device type. However, it does not include a vocabulary to describe more abstract widgets [123]. PUC describes device functions in terms of state variables and commands, with a grouping mechanism used for placement of UI objects. The INCITS/V2 URC standards define a generic framework and an XML-based user interface language to let a wide variety of devices act as a remote to control other devices, called targets.

*User interface remoting* uses a remote interface protocol that relays I/O events between an application and its user interface. The user interface resides on a remote platform instead of on the device itself. The Universal Plug and Play (UPnP) Remote User Interface (RUI), that forms part of the UPnP AV standard, belong to this category. UPnP RUI

follows the Web server-client model, where the controller acts as a remote user interface client, and the target, acting as a remote user interface server, exposes a set of user interfaces [105].

CEA-2014, that builds on the [UPnP RUI](#) interface, uses a matchmaking process for a controller device to select a user interface protocol that is supported by the controller platform [123]. Supported protocols include AT&T Virtual Network Computing ([VNC](#)) and Microsoft Remote Desktop Protocol ([RDP](#)). [VNC](#) uses the Remote Framebuffer ([RFB](#)) protocol to send pixels and event messages between devices.

Universal UI languages and UI remoting are orthogonal approaches [62]. UI remoting might be used in parallel with device-independent UI languages.

In this thesis we are more interested in tangible interactions in ubiquitous computing environments, instead of the usual [GUI](#)-based solutions. Smart environments need not only descriptions of [GUI](#)-based input/output, but also of the physical input/output capabilities, hardware capabilities, network capabilities and other characteristics of devices. The first attempt to define a vocabulary that conveys these device characteristics was the W3C Composite Capabilities/Preferences Profile ([CC/PP](#)). Other approaches to describe device characteristics that are not [GUI](#) specific are described in the next section.

*Current remote user interfaces are device-oriented rather than task-oriented. CEA-2018, discussed earlier in Section 2.4.1, tries to solve this problem by using task model representations.*

## 7.1 RELATED WORK

### 7.1.1 UAProf

The WAP Forum's User Agent Profile ([UAProf](#)) specification is an Resource Description Framework ([RDF](#))-based schema for representing information about device capabilities. UAProf is used to describe the capabilities of mobile devices, and distinguishes between hardware and software components for devices.

For example, in the Nokia 5800 XpressMusic UAProf profile<sup>1</sup>, its interaction capabilities are described as follows:

- PhoneKeyPad as Keyboard
- 2 as NumberOfSoftKeys
- 18 as BitsPerPixel
- 360x640 as ScreenSize
- Stereo as AudioChannel

Other user interaction capabilities are defined in a Boolean fashion of yes/no, e.g. `SoundOutputCapable`, `TextInputCapable`, `VoiceInputCapable`.

*W3C's [CC/PP](#) is also an [RDF](#)-based schema.*

<sup>1</sup> [nds1.nds.nokia.com/uaprof/Nokia5800d-1r100-2G.xml](http://nds1.nds.nokia.com/uaprof/Nokia5800d-1r100-2G.xml)



### 7.1.2 Universal Plug and Play (UPnP)

UPnP with its device control protocols is one of the more successful solutions<sup>2</sup> to describing device capabilities. However, it has no task decomposition hierarchy and only allows for the definition of one level of tasks [73].

UPnP was developed to support addressing, discovery, eventing and presentation between devices in a home network, and the current version (1.1) was released as the ISO/IEC 29341 standard in 2008. It consists of a number of standardised Device Control Protocols (DCPs) - data models that describe certain types of devices. The DCPs that have been adopted by industry include audio/video, networking and printers. DCPs for low power and home automation have not yet been adopted.

Digital Living Network Alliance (DLNA) is a complete protocol set around Internet Protocol (IP) and UPnP, where to be certified for DLNA, a device needs to have UPnP certification first. This protocol set was developed mainly to increase interoperability between Audio/Video (AV) equipment in the home. It achieves this by limiting the amount of options available in the original protocol standards.

When describing the capabilities of a smart object, not only the interaction capabilities are important, but also the device states. With UPnP, two types of documents are used to describe device capabilities and states. A *device description document* describes the static properties of the device, such as the manufacturer and serial number [56]. UPnP describes the services that a device provides in *service description documents*. These XML-based documents specify the supported actions (remote function calls) for the service and the state variables contained in the service.

The state variable descriptions are defined in a similar way to how we define our interaction primitives, with a unique name, required data type, optional default value and recommended allowed value range. The UPnP Forum has defined their own custom set of data types, with some similarity to the XML Schema data types used by Web Ontology Language (OWL) 2. As an example, consider a state variable to describe the darkness of a piece of toast, where `ui1` is defined as an unsigned 1-byte integer:

```
<stateVariable sendEvents="no" >
  <name>darkness</name>
  <dataType>ui1</dataType>
  <defaultValue>3</defaultValue>
  <allowedValueRange>
    <minimum>1</minimum>
    <maximum>5</maximum>
```

<sup>2</sup> <http://upnp.org/sdcpss-and-certification/standards/sdcpss/>

*In an IEEE ComSoc online tutorial entitled Consumer Networking Standardizations, Frank den Hartog from TNO stated that "DLNA has been a major effort to get computer people to talk to consumer electronics people".*

```

        <step>1</step>
    </allowedValueRange>
</stateVariable>

```

The `sendEvents` attribute is required for all state variable descriptions. If set to "yes", the service sends events when it changes value. Event notifications are sent in the body of an HTTP message and contain the names and values of the state variables in XML.

Let us consider these device states in terms of user interaction. There are four key concepts in an interaction - actions, states (internal to the device), indicators, and modes (physically perceivable device states) [101]. The user performs actions, which change the device state, which in turn control indicators (augmented feedback). Users may not know exactly which state or mode a system is in. If we want to fully capture the capabilities of the device, we need to specify the device states, the transitions between these states, the interaction primitives which can cause these state changes, as well as the default and current states of the device. When this device is then connected to another device, we also need a way to communicate state changes to the other device.

*Our approach to modelling devices states and state transitions using Finite State Machines (FSMs) is described in Section 6.5.*

### 7.1.3 SPICE DCS

The SPICE Mobile Ontology<sup>3</sup> allows for the definition of device capabilities in a sub-ontology called Distributed Communication Sphere (DCS) [114]. A distinction is made between device capabilities, modality capabilities and network capabilities. While the ontology provides for a detailed description of the different modality capabilities, e.g. being able to describe force feedback as a `TactileOutputModalityCapability`, there are no subclass assertions made for other device capabilities. Most physical characteristics of the devices are described via their modality capabilities, e.g. a `screenHeight` data property extends the `VisualModalityCapability` with an integer value, and the `audioChannels` data property is also related to an integer value with `AcousticModalityCapability`. The input format of audio content is described via the `AcousticInputModalityCapability` through an `inputFormat` data property to a string value.

It is not clear whether the modality capabilities should be used to describe the actual content that may be exchanged or the user interaction capabilities. As an example, if a device has an `AcousticOutputModalityCapability`, it is not clear whether the device can provide user interaction feedback (e.g. in the form of computer-generated speech or an audible click), or that the device has a speaker.

<sup>3</sup> <http://ontology.ist-spice.org/>

## 7.2 REGISTERING DEVICES ON STARTUP

Based on this existing work, we now look at how their functionalities should be registered, as well as how devices can be identified in the digital and physical domain.

On device startup, the smart object registers its digital and physical identification information (e.g. RFID tag or IP address) and its functionality with the SIB, and then subscribes to new connections and events as shown in Figure 43.

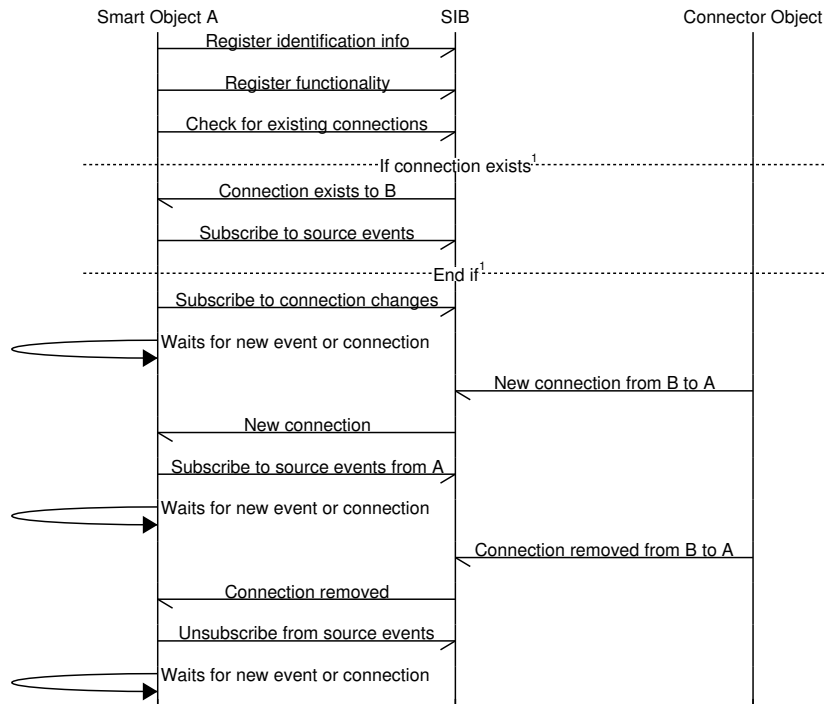


Figure 43: Startup sequence between smart object and SIB

This sequence is the same for all smart objects that connect to the Semantic Information Broker (SIB), and should be implemented in every Knowledge Processor (KP) that uses our approach. You might notice some parallels between the SIB concept and the Microsoft Windows Registry. The Registry is used to store configuration information of software applications on a single device, while the SIB is used (among other things) to store device functionality descriptions of a system of devices. However, compared to the Windows Registry, which is a basic hierarchical key-value store, the triple store and reasoning engine used in the SIB provide a number of advantages, including subsumption testing, consistency checking and the ability to use restrictions to constrain data instances. This means we can use reasoning to verify the consistency and stability of the data in the SIB.

We now discuss the first two steps of the sequence diagram in Figure 43, registering identification and functionality, in more detail.

*The startup sequence contains instances of the blackboard and publish/subscribe patterns described in Section 10.2.*

*Subsumption testing, consistency checking and restrictions are discussed in more detail in Section 9.1.*

### 7.2.1 *Identifying devices*

In order to discover a device's capabilities, it is first necessary to be able to uniquely identify the device. Today it is common to identify groups of products using barcodes and other numbering systems. Before ubiquitous computing, only expensive things such as precious metals, currency, or large machines were individually identified with any regularity [59]. New tracking technologies like Radio Frequency Identification (RFID) tags and smart cards allow us to link a unique identification number to a specific physical product, like a smart phone that identifies a specific person to the phone network. IPv6, an extension to the Internet Protocol standard, allows us to identify approximately  $3.4 \times 10^{38}$  objects in the digital domain.

Mavrommati et al [66] linked an XML-based description of an object's properties, services and capabilities with an artefact ID. This alphanumeric ID is mapped to a namespace-based identification scheme, using a similar process to the one used for computer MAC addresses.

Tungare et al [102] identified an information object in their Syn-cables framework, used to migrate task data and state information across platforms, via a Uniform Resource Identifier (URI). They used the structure

```
sync://<info-cluster-id>/<collection>/<type>/<path>/<object-name>
```

where the information cluster is the set of all devices a user interacts with during the course of a day. Each of the devices in an information cluster "offers a unique set of affordances in terms of processing capabilities, storage capacities, mobility constraints, user interface metaphors, and application formats".

Most service discovery mechanisms, for example those used by UPnP, assume the user will use the Internet to establish connections [56]. However, when we are in close proximity to things, we can address these things by pointing at them, touching them or by standing near them, instead of having to search or select them through a GUI.

Olsen et al. [81] used the domain name or IP address of a software client associated with a device to identify it, and a URL to identify services associated with a specific device. A user was associated with a URL used for that user's current session. The physical user was identified using a Java ring, with a small Java virtual machine running on the ring's microcontroller.

O'Reilly and Battelle [82] argue that formal systems for adding a priori meaning to digital data are actually less powerful than informal systems that extract that meaning by feature recognition. They think that we will get to an Internet of Things via a "hodgepodge of sensor data, contributing, bottom-up to machine-learning applications that gradually make more and more sense of the data that is handed to them". As an example, consider that using smart meter data to extract

a device's unique energy signature, it is possible to determine the make and model of each major appliance.

Jeff Jonas's work on *identity resolution* uses algorithms that semantically reconcile identities [94]. His Non-Obvious Relationship Awareness (NORA) technology is a *semantically reconciled and relationship-aware directory* that is used by the Las Vegas gaming industry to identify cheating players within existing records. A semantically reconciled directory recognises when a newly reported entity references a previously observed entity.

We agree that waiting until every object has a unique identifier for the Internet of Things to work is futile. However, the Semantic Web was designed with this problem in mind. We can use a [URI](#) to identify an entity we are talking about. Different people will use different [URIs](#) to describe the same entity. We cannot assume that just because two [URIs](#) are distinct, they refer to the same entity [4]. This feature of the Semantic Web is called, the *Non-unique Naming Assumption*. When using [OWL](#), it is necessary to assert that individuals are unique using the `owl:allDifferent` or `owl:differentFrom` elements. Individuals can be inferred to be the same, or asserted using `owl:sameAs`. For [OWL](#) classes and properties, we can use `owl:equivalentClass` and `owl:equivalentProperty`.

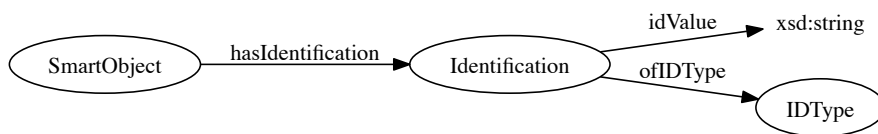


Figure 44: Modelling identification in the ontology

We modelled the identification of smart objects as shown in Figure 44. An example of how the Squeezebox [KP](#) can be linked to both its IP address and [RFID](#) tag is shown below:

```

SqueezeboxKP a SmartObject .
SqueezeboxKP hasIdentification id1234 .
SqueezeboxKP hasIdentification id4567 .
id1234 ofIDType IPAddress .
id1234 idValue "192.168.1.4:1234" .
id4567 ofIDType RFID_Mifare .
id4567 idValue "12AB45CD67EF" .
  
```

### 7.2.2 Registering a device's functionality

In the first design iteration we used a very simple approach to modelling the capabilities of devices, where provides and consumes properties linked smart objects to the names of the capabilities. During the later design iterations we modelled capabilities as functionalities of a device instead.

*Examples of provides and consumes were shown in Section 3.2.*

To register the functionality of a device such as the Squeezebox internet radio, we can use the following triples:

```
squeezeboxKP a SmartObject .
squeezeboxKP functionalitySource Alarm .
squeezeboxKP functionalitySink Alarm .
squeezeboxKP functionalitySink Music .
```

This indicates that the device is capable of acting both as a source and as a sink for Alarm functionality, while it can act as a sink for Music functionality. Once these device capabilities are registered, we can use semantic reasoning to infer which devices can be connected to each other.

## 7.3 REASONING WITH DEVICE CAPABILITIES

A smart object can have one or more functionalities that can be shared with other smart objects. As shown in the previous section, we model a functionality as

```
ie:Alarm a ie:Functionality .
ie:phone1 a ie:SmartObject .
ie:phone1 ie:functionalitySource ie:Alarm .
```

or in the case of modelling the functionality of a sink we use

```
ie:Music a ie:Functionality .
ie:speaker1 a ie:SmartObject .
ie:speaker1 ie:functionalitySink ie:Music .
```

To infer that two devices can be connected based on functionality as shown in Figure 45, we use an OWL 2 property chain:

$$\text{functionalitySource} \circ \text{isFunctionalityOfSink} \sqsubseteq \text{canConnectTo}$$

where we use `isFunctionalityOfSink`, the inverse property of `functionalitySink`, to be able to create the property chain.

To prevent a smart object from having a `canConnectTo` relationship to itself (which will be the case for semantic transformers), the relationship is defined to be irreflexive. Inferring indirect connection possibilities is also possible with a property chain:

$$\text{canConnectTo} \circ \text{canConnectTo} \sqsubseteq \text{canIndirectlyConnectTo}$$

*A similar method was used to match media types in Section 4.4.2.*

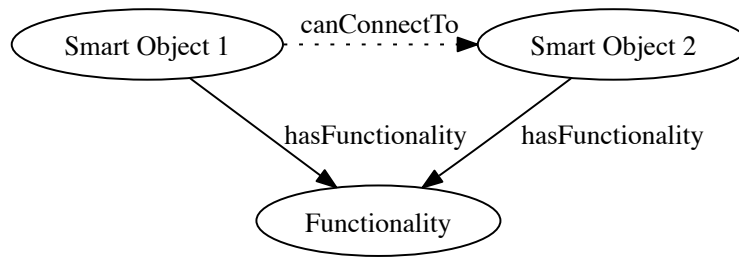


Figure 45: Inferring connection possibilities based on functionality

### 7.3.1 Representing functionalities as predicates

If we want to model the common functionalities between two smart objects, we can use the n-ary ontology design pattern [78]. Unfortunately, this is not intuitively readable from its ontological representation, as shown in the top half of Figure 46. The representation looks complicated and is difficult to read. On the other hand, we can also directly infer the matched functionalities as predicates, instead of using n-ary representations. The result can be represented using three triples instead of nine triples, and it is also more intuitively understandable, as shown in the bottom half of Figure 46.

*Ontology design patterns are discussed in Chapter 9.*

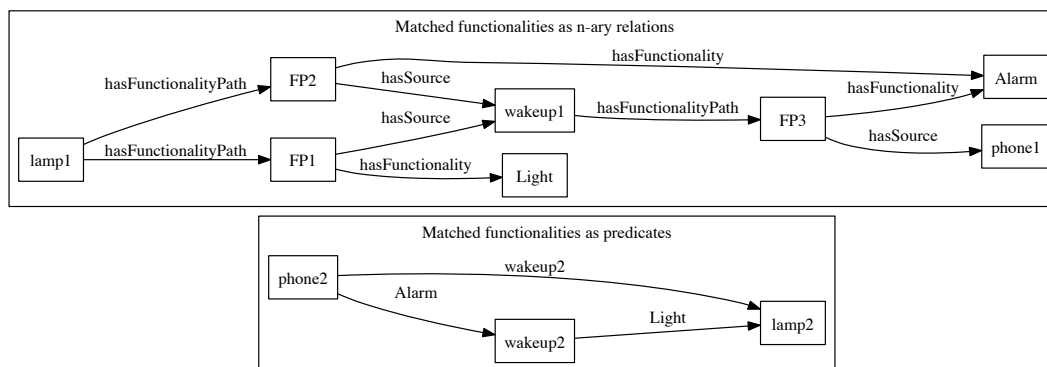


Figure 46: Representing matched functionalities: N-ary relations versus predicates

Representing an individual as a predicate is not valid in OWL 2 DL and places the ontology into OWL 2 Full. However, since we are using an OWL 2 RL/RDF Rules reasoning mechanism, this is not an issue. Thus we choose to use predicates instead of n-ary relations, and so

we do not stay within OWL 2 DL. We can easily infer this relation using a SPARQL Inferencing Notation (SPIN) rule:

```
CONSTRUCT{
    ?this ?functionality ?sink .
}
WHERE{
    ?this :functionalitySource ?functionality .
    ?sink :functionalitySink ?functionality .
}
```

where `?this`  $\equiv$  `SmartObject`. For example, if we have functionalities defined as

```
:phone1 :functionalitySource :Music .
:speaker1 :functionalitySink :Music .
```

the above SPIN rule will infer

```
:phone1 :Alarm :speaker1 .
```

such that the functionality is represented as a predicate. For a semantic transformer, which is indirectly connected to smart objects, we need an additional SPIN rule:

```
CONSTRUCT{
    ?source ?this ?sink .
}
WHERE{
    ?source :canIndirectlyConnectTo ?this .
    ?this :canIndirectlyConnectTo ?sink .
}
```

where `?this`  $\equiv$  `SemanticTransformer`. This infers the semantic transformer itself as the relation between the source and the sink, since it transforms the original functionalities. For example, using the smart objects in Figure 46, if we have

```
:phone2 :functionalitySource :Alarm .
:wakeup2 :functionalitySink Alarm .
:wakeup2 :functionalitySource :Light .
:lamp2 :functionalitySink :Light .
```

and the two property chains from Section 7.3, we can infer that

```
:phone2 :canConnectTo :wakeup2 .
:wakeup2 :canConnectTo :lamp2 .
:phone2 :canIndirectlyConnectTo lamp2 .
```



Using the [SPIN](#) rule defined above we can then infer that

**:phone2** :wakeup2 **:lamp2** .

where the semantic transformer itself becomes the predicate between the two smart objects, signifying the possibility of having wakeup service functionality between the two objects.

How can we provide feedback to the user that these possible functionalities exist between the smart objects? This can be done using the Connector object and the feedback capabilities of the devices themselves. Just after the user scans the second device and before the connection is actually made, feedback of the different possibilities for shared functionality can be provided to the user.

When two devices can be connected directly, the Connector object creates a temporary connection from itself to the sink, and generates a PreviewEvent with the matching functionality as dataValue. The sink creates a preview of the functionality based on the PreviewEvent and its dataValue. When the sink completes the preview, it generates its own PreviewEvent to indicate this. The Connector object sees the sink's PreviewEvent and removes the temporary connection.

However, when there is a semantic transformer between the source and the sink, the Connector object creates a temporary connection to the semantic transformer instead of the sink, in order to generate the appropriate PreviewEvent, as shown in Figure 47. This temporary connection is specified using the tempConnectedTo property, a sub-property of the connectedTo property.

*The dataValue property of interaction events is discussed in more detail in Section 8.2.*

*Preview events and the tempConnectedTo property were first discussed in Section 5.4.1.1.*

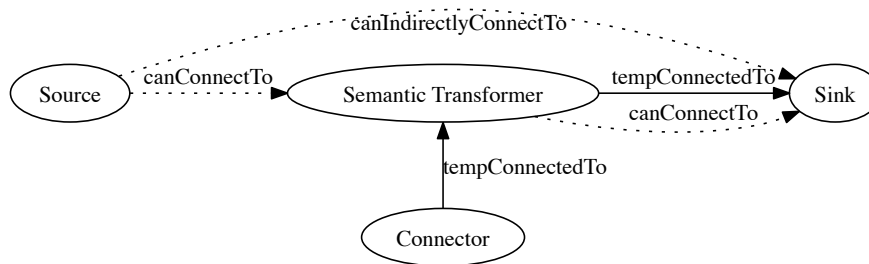


Figure 47: Temporary connections for PreviewEvent when semantic transformer is used

The Connector object uses the inferred canIndirectlyConnectTo and canConnectTo properties to determine where to insert the tempConnectedTo properties. After inserting the properties, the Connector object generates a PreviewEvent. Due to the tempConnectedTo relationship between the Connector object and the semantic trans-

former, the semantic transformer responds to this event and also generates a `PreviewEvent` with its functionality as `dataValue`. Due to the `tempConnectedTo` relationship between the semantic transformer and the sink, the sink responds to the preview event and generates the appropriate preview of its functionality.

Now that we have a way to model the capabilities of devices and provide previews of their functionality, we can start looking at ways to represent the different kinds of events that are generated on these devices in the next chapter.

## Part IV

### APPENDIX

