XSED – XML-Based Description of Status–Event Components and Systems

Alan Dix[1], Jair Leite[2], and Adrian Friday[1]

[1] Computing Department, Lancaster University, Infolab21, South Drive, LA1 4WA, UK
[2] Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Lagoa Nova, 59078-970, Natal, RN, Brazil
alan@hcibook.com, jair@dimap.ufrn.br, adrian@comp.lancs.ac.uk
http://www.hcibook.com/alan/papers/EIS-DSVIS-XSED-2007/

Abstract. Most user interfaces and ubiquitous systems are built around event-based paradigms. Previous work has argued that interfaces, especially those heavily depending on context or continuous data from sensors, should also give attention to status phenomena – that is continuously available signals and state. Focusing on both status and event phenomena has advantages in terms of adequacy of description and efficiency of execution. This paper describes a collection of XML-based specification notations (called XSED) for describing, implementing and optimising systems that take account of this dual status–event nature of the real world. These notations cover individual components, system configuration, and separated temporal annotations. Our work also presents a implementation to generate Status-Event Components that can run in a stand-alone test environment. They can also be wrapped into a Java Bean to interoperate with other software infrastructure, particularly the ECT platform.

Keywords: Status–event analysis, reflective dialogue notation, ubiquitous computing infrastructure, XML, temporal properties.

1 Introduction

This paper describes a collection of XML-based specification notations for describing, implementing and optimising status–event based systems. The notations are collectively called XSED (*pron. exceed*) – XML Status–Event Description.

User interfaces are nearly universally programmed using an event-based paradigm. This undoubtedly matches the underlying computational mechanism and is thus necessarily the way the low-level implementation deals with execution. However, this event-based paradigm is also evident in the way in which interfaces are described at a higher-level and this is more problematic. This purely event-oriented view of interfaces has been critiqued for a number of years and status–event analysis proposes a view of interaction that treats status phenomena (those that have some form of persistent value) on an equal footing with event phenomena [1,2]. For example, when

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-540-92698-6_37

J. Gulliksen et al. (Eds.): EIS 2007, LNCS 4940, pp. 210–226, 2008.
© Springer-Verlag Berlin Heidelberg 2008

dragging a window the window location and mouse location are both status phenomena and the relationship between them should be described in terms of a continuous relationship over time.

Arguably the status-oriented interactions in a traditional GUI (principally mouse dragging and freehand drawing) are to a large extent the exception to the rule of more event-focused interaction (button click, key press). However, in ubiquitous or pervasive environment the reverse is often the case. Sensors tend to monitor status phenomena such as temperature, pressure, sound level. At a low level these sensor values are translated into discrete data samples at particular moments, but unlike the moment of a mouse click, the particular times of the samples are not special times, merely convenient ones to report at. So at a descriptive level it is inappropriate to regard them as 'events' even if they are implemented as such at low-level. Furthermore the data rates may be very high, perhaps thousands of samples per second, so that at a practical level not taking into account their nature as status phenomena can be critical for internal resource usage and external performance and behaviour.

In this paper we will examine some of these issues and present a collection of XML-based specification notations, XSED, for embedding full status–event processing within event architectures used in ubiquitous and mobile computing. Notations are included for individual components (encoded as an extension to the W3C XML finite state machine specification), configuration of multiple components and annotations of specifications for runtime optimisation. The notations together allow local and global analysis and run-time introspection. The specifications are transformed into Java code for execution and can be wrapped as Java Bean components for execution within the ECT infrastructure [3].

2 Status–Event Analysis

2.1 What Is It?

Status–event analysis is concerned with the issues that arise when you take into account the distinction between status and event phenomena. The distinction is quite simple:

events – things that *happen* at a particular moment: mouse click, alarm clock rings, thunder clap

status – things that *are* or in other words always have some value that could be sampled: screen contents, mouse location, temperature, current time or weather

Note that the word 'status' is used rather than 'state' because of the connotations of internal state in computer systems. Whilst this internal state is an example of a status, status also includes things like the temperature, patterns of reflected light, average walking speed of a crowd.

Note too that status phenomena may be continuous (temperature) or discrete (is the light on) – the critical thing is their temporal continuity. Figure 1 demonstrates this. Status phenomenon labelled (1) has a continuously varying value over time, but the status phenomenon (2) has a number of discrete values, but still at any moment has a well defined value (except possibly at moments of transition). In contrast the event phenomena (3) and (4) occur only at specific times. The two event phenomena (3) and (4) are also shown to demonstrate that event phenomena may be periodic (3) or irregular (4).

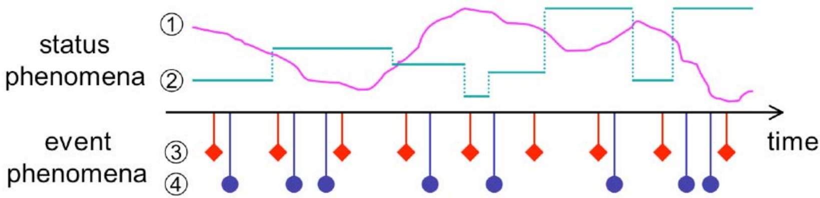

Fig. 1. Status and event phenomena over time

Status–event analysis has proved a useful way to look at interactive systems because it is able to describe phenomena that occur in human–computer interactions, in human– human interactions, in human interactions with the natural world, and in internal computational processes. For example, one way in which an active agent can discover when a status phenomena has changed is to poll it; this occurs internally in a computer, but also at a human level when you glance at your watch periodically in order not to miss an appointment. As an internal computational paradigm it has also used in commercial development (see section 2.3 below).

Status–event analysis draws its analytic power not just from establishing distinctions, but also from the inter-relationships between status and event phenomena, like the polling behaviour above. Clearly events may change status phenomena (e.g. turning on a light) and a change in status may be noticed by an agent and become a *status change event*. Events may give rise to other events as in event-based systems, but also a status may depend on one or more other status, for example the window location tracking the mouse location – a *status–status mapping*.

2.2 Does It Matter?

Whilst there are clearly real distinctions between classes of phenomena, is it important that these are reflected in specification and implementation of systems? In fact there are a number of reasons why it is important to explicitly encode both status and event phenomena in specifications.

Purity and Capture. The first reason is just that it is right! This is not just a matter of theoretical purity, but of practical importance. The most costly mistakes in any development process are those made at requirements capture. Describing things in the way in which they naturally are is more likely to lead to correctly formulated requirements. Whilst later more formal manipulations (whether by hand or automated such as compilers) can safely manipulate this, but the initial human capture wants to be as natural as possible. For example, compare a status-oriented description: "Record a meeting when Alison and Brian are both in the room"; with an event-oriented one: "Record a meeting when there has been an ‘Alison Enters’ event followed by events not including ‘Alison Leaves’ followed by ‘Brian Enters’ OR a ‘Brian Enters’ followed by events not including ‘Brian Leaves’ followed by ‘Alison Enters’". Which is easier to understand and more likely to be expressed correctly?

Premature Commitment. Even if one captures a required behaviour correctly the conversion of status–status mappings to event behaviours usually involves some form of ‘how’ description of the order in which lower level events interact in order to give higher

level behaviour. That is a form of premature commitment. An example of this occurred recently in the development of a visualisation system. A slider controlled a visualisation parameter. This was encoded by making each slider change event alter the underlying data structures, creating a storm of data updates and screen repaints – (prematurely committed) event-based specification of what is really a status–status mapping.

Performance and Correctness. We have seen how the lack of explicit status–status mappings can lead to interaction failure! If the system 'knew' the relationship between slider value and visualisation appearance, it could infer that updates to the internal data structures are only required when a repaint is about to happen. In general, this lack of explicit status knowledge can lead to both local computational resource problems and also excessive network load in distributed systems. In a sensor-rich system with high-data-rate sensors this is critical. Typically this is managed on an ad hoc basis by throttling sensors based on assumed required feed rate. However, this does not allow for dynamic intervention by the infrastructure if the system does not behave in the desired fashion, for example, to modify the rate of event sources.

An example of this occurred in 'Can you see me now' a ubiquitous/mobile game [4]. The location of each player was shown on a small map, but during play the locations of players lagged further and further behind their actual locations. The reason for this turned out to be that the GPS sensors were returning data faster than it was being processed. The resulting queue of unprocessed events grew during the game! Clearly what was wanted was not that for every GPS reading there was a corresponding change on the map (an event relationship), but instead that the location on the map continually reflected, as nearly as possible, the current GPS location (a status–status mapping).

The phrase "as nearly as possible" above is important as any status–status mapping inevitably has delays, which mean it is rarely completely accurate. For simple mappings this simply means small lags between different status–status phenomena. However, if different status phenomena have different lags then incorrect inferences can be made about their relationships [5]. For example, on a hot summer day if the house gets warmer but sensors in the hotter part of the house have longer lags than the cooler parts, then a climate control system may set fans to channel air the wrong way. In such cases explicit encoding of the status–status mapping would not remove the inherent problem of sensing delays, but would make the interdependency apparent and allow setting of parameters such as expected/required jitter between sources, forms of generalised 'debounce' etc.

2.3 Existing Status–Event Systems/Notations

A notation was used in [2] for status–event analysis, in particular allowing the specification of interstitial behaviours of interfaces – the status–status relationships that occur in-between major events, which are often what gives to an interface its sense of dynamic feel. This was targeted purely at specification and theoretical analysis although is potentially not far from an executable form. Some other specification notations, whilst not based on status–event analysis, embody aspects of status phenomena. Wüthrich made use of cybernetic systems theory with equations close to those in continuous mathematics to describe hybrid event/status systems [6] and different forms of hybrid Petri Nets have been used by different authors [7,8]. Also there is a whole sub-field of formal methods dedicated to hybrid systems specification although principally focused

on mixing continuous external real world behaviour with discrete computational behaviour [9].

Further theoretical work on event propagation in mixed status–event systems showed the importance of a strong distinction between data flow direction and initiative [10]. Often status–status mappings are better represented at a lower level by demand-driven rather than data-driven event propagation. The Qbit component infrastructure in the commercial onCue system were constructed to enable this flexibility [11]. Each Qbit may have ‘nodes’ (like Java Bean properties) of various kinds. Of the unidirectional nodes, there are familiar *get* and *set* nodes where a value is output or input under external control, *listen* nodes where a value can be output to a listener under internal control and finally *supply* nodes that allow the Qbit to request a value from an unknown external source. The last, that naturally completes the space of (single directional) node types, is particularly important as it allows demand-driven data flows with external connections.

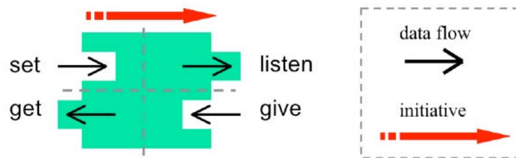

Fig. 2. Qbit nodes

Note however, that the Qbit component still does not represent status phenomena explicitly; instead it aims to make event representations easier and more flexible. The onCue product it supported was a form of context-aware internet toolbar, so shared many features with more physical sensor-based systems. Another critical feature of the Qbit framework that we have preserved in XSED, is *external binding* – because of the symmetry of the Qbit I/O model it is possible to wire up Qbits without 'telling' the Qbit what it is connected to. In comparison the listener model used for Java Beans requires each Bean to 'remember' what has asked to be told about changes.

3 The XSED Notation

In order to better represent both kinds of phenomena for ubiquitous interactions we have defined and implemented an executable description notation XSED that explicitly encodes status and event phenomena. The description notation XSED includes four key elements:

1. Individual software components – descriptions that include both status and event input and output, and specification of the mappings between input and output
2. Configuration – showing, without reference to specific components, how several components fit together architecturally to make a lager component
3. Binding – filling the component 'holes' in a configuration description with specific components
4. Annotation – additional timing and other information referring to the configuration links that can improve the performance of a system.

The separation of components allows a level of reuse and 'plug and play' between components. For example, an infra-red motion sensor may be used by both a burglar alarm system and to adjust the lighting when people leave a room. The separation of binding from configuration allows the same flexibility for the system as a whole: the alarm could use ultrasound instead of infra-red motion detection. The separate annotation allows varying levels of designer or automated analysis to inform optimisations of the system. For example, a temperature sensor may be able to deliver thousands of readings a second, but we may only require the temperature once per second. The knowledge of the appropriate sensor rate depends on the particular set of components, their configuration and particular external constraints in and information (e.g. the maximum rate at which temperatures change in the environment). This information does not belong in individual components, nor the configuration, nor the binding. In addition, separating performance-oriented annotation allows a level of plug-and-play for analytic tools as it gives a way for a static or dynamic analysis of a system to be fed into its (re)construction.

The concrete syntax is in XML for structural description with embedded JavaScript for computational elements. While this choice can be debated it follows successful XML-based notations such as XUL [12] and could easily be generated as intermediate form by other forms of graphical or textual notation. The XML notation can be used for design-time analysis, executed through an interpreter directly, or transformed into Java for integration with other software.

4 Individual Components

The individual components in XSED may represent individual UI widgets, sensors interfaces, or more computational processing. The notation follows [2] in declaring explicit status input and output as well as event input and output. One aim is to make a description that is easy for a reflective infrastructure to analyse hence we have chosen initially to encode the internal state of each component as a finite state machine rather than to have arbitrary variables in the state as in [2]. Other aspects of the notation (configuration, binding and annotation) do not depend on this decision. So it is independent, with limited repercussions. We can therefore revisit this decision later if the capabilities of the FSM are too restrictive, but it initially allows easier analysis. The use of a FSM also parallels the early Cambridge Event architecture to allow comparison between solely event-based and status–event descriptions. For concrete syntax we use XML extending the W3C draft standard for finite state machines [13].

4.1 XML Specification

Figure 3 show the top level syntax of a single Status-Event component (see also web listings 1 and 2). A single SE component has initial input and output declarations each of which may be a status or event. This may be followed by default status–status mappings giving output status in terms of input status. The states also contain status–status mappings (the defaults mean that these can be incomplete otherwise every output status must be given a value for every state). Finally the transitions describe the effects of events in each state. Each transition has a single input event that triggers the transition and a condition. As well as causing a change of state may also cause output events to fire.

```
xmlspec ::= input output defaults state* transition*
input ::= ( status | event )*
output ::= ( status | event )*
defaults ::= status-out*
state ::= id {start} status-out*
transition ::= state-ids event-in {condition} event-out*
```

Fig. 3. Overall structure of XSED specification

The status–status mappings (status-out) and event outputs have values given by expressions and the transitions conditional is a boolean expression. These can only access appropriate input status/events. In the case of output status, only the input status can be used as there are no events active. In the case of transition conditions and event outputs the value (if there is one) of the triggering event can also be used in the expressions.

4.2 Transforming and Executing the Specification

The XML specification is parsed into an internal Java structure. This follows a four stage process:

1. *Parsing* – the XML specification is read into an internal DOM structure using standard XML parsing
2. *Marshalling* – the XML DOM is transformed into a bespoke internal structure that still represents the specification, but does so in dedicated terms specialised methods etc. Note that this level of representation would be shared with any alternative concrete syntax. This level of representation is also suitable for static analysis.
3. *Building* – the specification is used to construct data structures and Java code suitable for execution. The components generated can optionally be wrapped as a Java Bean suitable for embedding in other execution environments, notably ECT [3]. Note, some elements of the specification are retained at runtime in the component schema to allow runtime reflection used during configuration linkage.
4. *Running* – the generated Java code is compiled and placed in suitable folders, Jar files, etc. for deployment either in EQUIP or in stand-alone test environment.

The component generated from the XML description during the *build* phase implements a generic interface that is neutral as to the precise notation used. This is so that we can experiment with different kinds of status–event 'savvy' notations such as augmented process algebras. Figure 4 shows this interface. It provides methods to get the value of a specific named status of a status-event component and to fire a named event with a value. It is also possible to get the names and types (schema) of input and output status and events. The last method sets the environment to which the component interact with. Note that the `getStatus` method is for the *output* status and the `fireEvent` method is for *input* events.

```
public interface SEComponent {
    Object getStatus(String name);
    void fireEvent(String name, Object value);
    public Schema getSchema();
    public void setEnvironment(SEEnvironment environment);
}
```

Fig. 4. Abstract status–event component

The remaining two methods are (i) a reflection method `getSchema` that retrieves the names, types etc. of the inputs and outputs to allow dynamic binding and (ii) `setEnvironment` that gives the component a handle into the environment in which it operates. This Java environment object acts as a form of single callback where the component goes when it needs to access status input or to fire an event output.

The Java interface for `setEnvironment` is shown in Figure 5. As is evident this is exactly the same as the basic part of a component. That is for most purposes a component and the environment in which it acts are identical. This is not just an accident of the Java implementation but reflects the underlying semantics – one of the strengths of status– event descriptions is that it offers a symmetric description of interactions across a component boundary.

```
public interface SEEnvironment {
    Object getStatus(String name);
    void fireEvent(String name, Object value);
}
```

Fig. 5. Abstract status–event environment component

5 Configuration

As noted we separate out the configuration of components, what links to what, from the individual component specifications. This allows components to be reused in different contexts. Furthermore, while this configuration will typically be designed with particular components in mind, it is defined only in terms of schemas of expected components. This means the configuration can also be reused with different variants of components, or with a different set of components with similar interrelationships.

At an abstract level the configuration specification is similar to the architectural specifications in Abowd et al [14], that is a collection of named component slots with typed input/output nodes and linkage between them. However, Abowd et al., in common with most configuration notations, do not fully abstract over the components and instead frame the architecture specification over specific components. In addition, the typing on our components includes their status/event nature.

In the XML configuration file, each component lists its inputs and outputs each with a status/event tag and type (see web listing 3 for an example). A <links> section specifies the connections between the components. Static checking verifies whether output nodes always connect to appropriately types input nodes (including their status/event nature). In addition, static checking verifies that every status input has exactly one incoming link, whilst other forms of input/output node can have one, several or no connections.

The other unusual aspect of the configuration is the way in which it is packaged as a component in its own right. Instead of specifying inputs and outputs of the configuration as a whole, a 'world' component is added within the configuration. This looks exactly like all other components and represents the external environment of the set of components. This takes advantage of the fact, noted previously, that the status–event semantics are symmetric with respect to the environment – the environment of a component looks similar to the component as the component looks to the environment. When the configuration is bound with specific components it then becomes a single component that can be placed elsewhere. The interface of this aggregate component is precisely the dual of the world component – inputs to the aggregate component are effectively outputs of the environment and vice versa.

6 Binding

The actual binding of configuration to components is currently entirely within code at runtime. In order to link the SE components in a configuration, small proxy environments are produced for each component linked into the configuration. When a component request an input status, it asks its environment proxy, which then looks up the relevant source status in the internal representation of the component linkage. The relevant output status and component (linked to the requested input status) is then obtained. Similarly when an output event is fired this is passed to the proxy environment, which then finds the relevant input events on other components and fires these.

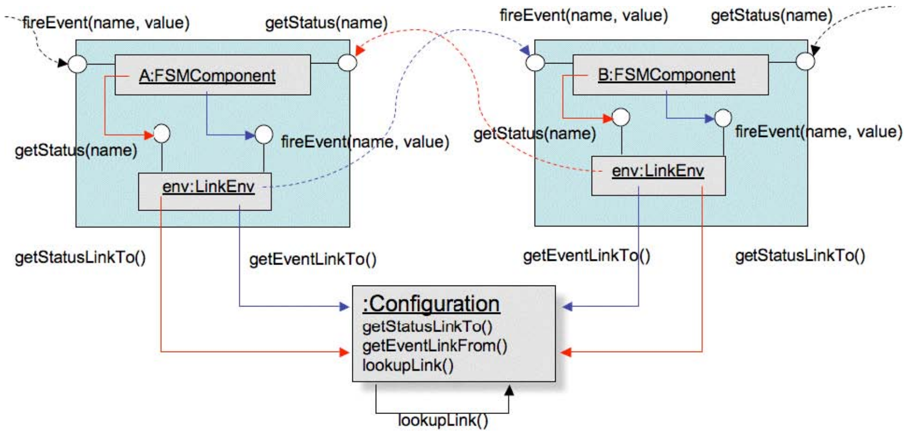

Fig. 6. Linking components

The link to the outside environment is produced using a special pair of dual SE components with the 'world' schema. These have no internal computation but simply reflect the inputs of one as the outputs of the other and vice versa. One end of the dual is bound into the configuration as the 'world' component and, once there, functions precisely like all other components within the configuration including having its own proxy environment. The other end of the dual pair then becomes the external view of the

aggregate component built form the configured components. Its inputs and outputs are then effectively the inputs and outputs of the configuration as a whole. This use of the dual radically simplifies the semantics of component aggregation.

7 Annotation

In the annotations description, unique link identifiers refer to links in a configuration file and specify properties of those links that can then be used to optimise the runtime behaviour of the system (see also web listing 4). The annotations include:

Initiative – Whether status links should be demand driven or data driven. Demand-driven status is the default behaviour where status inputs are requested when used. In contrast data-driven status is more similar to event propagation, where changes in status are pushed through the system. The latter is preferred if changes of status are rare compared to accesses.

Time – The timeliness of a link. A value of 1000 (milliseconds) means that data used can be up to 1 second 'out of date'. For a status link this would mean that if a status input is re-requested within 1 second of the last access the previous accessed value can be used. For an event link this means that multiple events within 1 second can be buffered and passed on together. For distributed system this can reduce overheads. This is similar to techniques used previously in the GtK (Getting-to-Know) notification server [15].

Last-or-all – When a sequence of events are fired whether all of them are important or just the last. In fact, when it is just the last, this is normally a sign that the event is a status change event. When this is combined with a timeliness annotation then multiple events within the specified time window can be suppressed and only the last one passed on.

Synchronisation – The timeliness annotations can mean that events and status change are not passed on in the same temporal order as they are produced. A synchronisation annotation over a collection of links specifies that order must be preserved over those links. For example, if an event is fired on one of the synchronised links then up-to-date status must be obtained for each of the synchronised status links no matter whether there is pre-accessed status of acceptable timeliness.

Note that these annotations may change the semantics as well as performance of the system. The production of the annotations, whether by a human designer or an automated global analysis, must ensure that this change in low-level semantics either does not change the higher-level semantics, or 'does not matter'. Such choices are always made in systems involving status phenomena as sampling rates are chosen depending on sensor, network or computational capacity or perceived required timeliness. However, normally these decisions are embedded deeply within the code or sensor choices, the separate annotation surfaces these decisions and separates these pragmatic decisions from the 'wiring' up of the components themselves.

As noted annotation is deliberately separated from the configuration as it will depend on the precise combination of components and the context in which they will be used. The separate annotation also means that the analysis tools (or had analysis) is separated from the use of the products of that analysis for optimisation during runtime.

8 Applying SE Components in Status/Event Infrastructures

We want to apply the generated Java Beans SE Components into existing distributed and ubiquitous infrastructures. We have chosen the ECT platform [3] because it supports events and states applying the concept of tuple spaces. In order to understand the requirements to support the status-event model and the advantages it can provides we present several computing architectures to deal with events and states.

8.1 Existing Status/Event Architectures

Over the last decade or so many researchers have attempted to design elegant and effective programming abstractions for building distributed systems. Space prohibits a full exploration here, but they can be loosely categorised as event based or state based.

8.1.1 Event Based Architectures

The Cambridge Event Architecture (CEA) is an example of an event based system [16]. In the context of their work with 'Active Badges', which allowed the tracking of electronic badge wearers throughout their research lab, the event architecture added the facility to build monitors that composed raw events together (e.g. Person A in room X, fire alarm activated etc.) to construct higher level information about the state of the world (e.g. fire alarm activated then person A left the building).

CEA was constructed using finite state machines composed of directed acyclic graphs of states ('beads'), representing start states, transitional states and final (accepting) states. Arcs could be standard (transition once when an event occurs) or spawning (create a new bead each time this transition occurs) – a spawning arc could be used to count every time a person left the building after the fire alarm, for example. Arcs may also be parameterised, which acts as a placeholder for information extracted from the state associated with each event (e.g. the badge holder's name). Handlers can be added to accepting states to trigger notifications to the applications deploying the monitors.

CEA provided an elegant declarative approach for specifying monitors and handlers. However, as the authors acknowledged in their paper, the order in which events occurred was sometimes hard to determine in the distributed case (meaning state machines would not transition correctly), moreover, it was not possible to represent the timely nature of events, nor whether events occurred within a certain time of each other – which can lead to unexpected generation of notifications. Most importantly for this discussion, while the system captures state internally (e.g. sensor identity, badge id) and can make this available to handlers, the status of the world must be actively reconstructed from monitoring events; if the monitor is offline when the event occurs, there is no facility to recover it.

In classic distributed systems, processes communicate with each other using virtual channels described by bindings and formed from pairs of endpoints (e.g. the well known BSD 4.3 Sockets API). Elvin, originating from DSTC [17], is an event broker that decouples application components in a distributed system. Producers form a connection to a broker and emit events (notifications) consisting of one or more typed name value pairs. Consumers connect to the Elvin broker and subscribe to particular

notifications matching a given regular expression. Subscriptions may express criteria for the name, type, value and conjunction of fields of interest within an event. The broker optimises the flow of matching events from producers to consumers based on the set of subscriptions it tracks. Events are propagated at best effort pace via the broker. A key advantage of this approach is that consumers may subscribe to events at any time, allowing for easy introspection of the internal communication between applications. Like CEA however, there is no persistence in the system so status cannot be reconstructed until the appropriate events are observed first hand by the consumer.

Brokers can be federated (also using the subscription language) to create larger distributed applications. The API lends itself to the creation applications based on the publication of content, such news tickers, chat applications and diagnostic monitors.

8.1.2 State Based Architectures

As a total contrast we also briefly consider state driven architectures. The classic example of such an approach is the canonical work by Gelernter [18] – Gelernter observed that coordinating the allocation of distributed computations in massively parallel computer architectures was I/O bound; much of the expected gains in computational throughput lost in the inter processor communication to coordinate the distribution of tasks to processors. The innovation in his approach was to build a computational model based around the generation and consumption of state in the form of typed tuples in an entity known as a 'tuple space'. A computational enhancement to existing programming languages (known as LINDA)[1] provided operations for adding, removing and observing content in the space. The task of allocating tasks to processors was turned from a producer driven model in which jobs were allocated to idle processors, to a consumer driven one in which idle processors pulled tuples (computations or part-computations) from the tuple-space and returned results to the space upon completion.

Since tuples persist in the tuple-space, producers and consumers do not have to be synchronously available to communicate – this is known as spatial and temporal decoupling. This feature of the paradigm has caused its adoption in mobile computing for dealing with loosely coupled distributed systems where parts of the application are seldom able to communicate synchronously [19,20].

As the tuple space paradigm has been used to build interactive systems it has become apparent that in order to support pacey interactions one must rapidly detect changes to the content of the tuple-space; something the original API was never designed for. Researchers have since augmented the standard API with additional operations (e.g. the EventHeap [21]) most notably to offer event notifications when tuples are added or removed [20]. Note that these operations are 'change of state' notifications on the tuple space, and do not support events between applications. If tuples reflect sensor values in the world then the tuple space may give us a historical view of status, but tuples do not necessarily reflect the current state of the world and it is left for the application writer to determine which information is the most current.

---

[1] Many researchers have since extended and explored alternate coordination based approaches; the interested reader is directed to Papadopoulos & Arbab [22].

8.1.3 Hybrid Event-State Architectures

The Equator Equip platform is[2] an example of a middleware that has drawn inspiration from the tuple-space concepts but, in addition, it includes support for passing events between applications. Equip 'dataspaces' contain typed objects that may contain state (as with the tuples in a tuple space). Different object types may be handled in different ways – objects that are of type 'event' trigger event notifications to a client application when they match an object representing a subscription to that type of event belonging to the application. This is, to our knowledge, the first system that attempts to offer an explicit separation between the use of the dataspace as a repository of shared information and the use of events for representing transient information (for example for tracking a user input device). By the taxonomy proposed by Papadopoulos and Arbab [22], Equip is a purely data-driven coordination model – all processes communicate via the dataspace; events are exchanged through the dataspace as specially typed objects.

The Equator Component Toolkit [3] (ECT) is a collection of java bean 'components' that provides a library of tools for constructing interactive ubiquitous computing applications. By linking ECT components together using a graphical editor designers can create limited 'interactive workflows' that are triggered by user interaction and interact with users in return via a range of physical prototyping and visualisation tools (such as Phidgets, MOTES, webcams etc). As components are dragged into the editor and as properties of the components are linked together to form a directed acyclic graphs, these get transformed into underlying objects, events and subscriptions in the Equip dataspace. Links between the properties of ECT components are stored as tuples in the shared Equip dataspace – note that, in contrast with channel based coordination models such as Reo [23], these links are application data as far as the dataspace is concerned and are not first class entities.

Distributed applications can be built by linking dataspaces together in client-server relationships or as synchronised peers. In equip, when two dataspaces link, historic events are explicitly regenerated by the system to bring both peers into exact synchronisation (a late joining client will see all events they would've seen had they been connected). When they disconnect, objects in a dataspace belonging to another peer are garbage collected. This behaviour has particularly interesting implications for interactive applications; the replay of historic state and events can appear as a fast motion replay of past activity, which is often meaningless or confusing for the user. Moreover, when a dataspace in a distributed application disconnects (potentially just because of a glitch in communications) and the data is garbage collected, the ECT components and connections between them that are represented are removed from the system (the application is partially destroyed). More importantly, the system would require a notion of which objects in the system represent status in the world and what the constraints are for their production and consumption to be able to optimise the flow of information between peers and application components. It is this issue we aim to explicitly address in XSED.

8.2 Generating SE Components to ECT Platform

Running the SE Components in the ECT platform [3] will enable us to use sensors, actuators and other components that have existing drivers/wrappers for ECT. In

[2] http://equip.sourceforge.net/

common with most ubicomp infrastructures, ECT is entirely event driven and this is achieved through listeners on Bean properties. Figure 9 show a SE Configuration Component (namely Config) running in ECT platform. The component is linked to three others ECT component (not generated by XSED) to illustrate our approach.

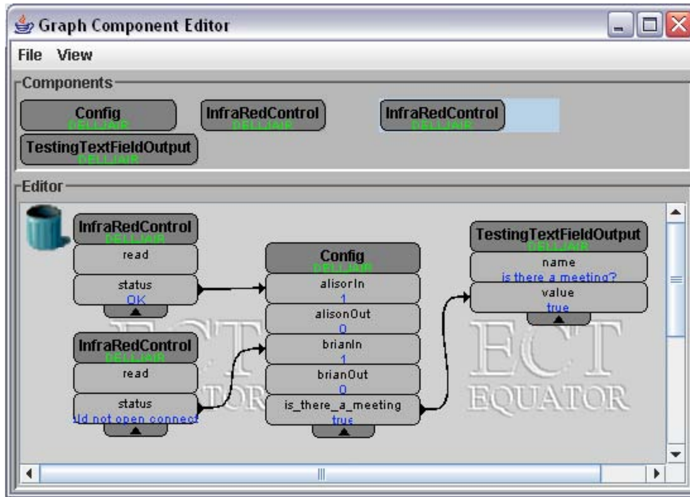

Fig. 9. A SE Configuration Component (Config) running in ECT platform

SE components are transformed into Beans using a wrapper class generated from the schema. For each input and output, both status and event, a Java slot is provided, but these are expected to be used differently depending on the type of node:

(i) event input – when the slot is set, the appropriate `fireEvent` is invoked.
(ii) status input – when the slot is set nothing happens immediately except the Bean variable being set, and when the component requires the status input (either when processing an event or when a status output is required), the variable is accessed.
(iii) event output – when the component fires the event the listeners for the relevant slot are called.
(iv) status output – when the `get`*`Name`* method is called for a slot, the corresponding status output is requested form the component (which may require accessing status input).

The 'wiring' in (i) and (iv) is directly coded into generated code for the Bean, but (ii) and (iii) require an environment for the component as the SE component simply 'asks' the environment for input status and tells it when an output event is fired. A proxy environment object is therefore also generated that turns requests from the component for status input into accesses on the internal Bean variables and when told that an output event has fired turns this into an invocation of the Bean change listeners. Figure 10 summarises these connections when an `FSMComponent` is wrapped.

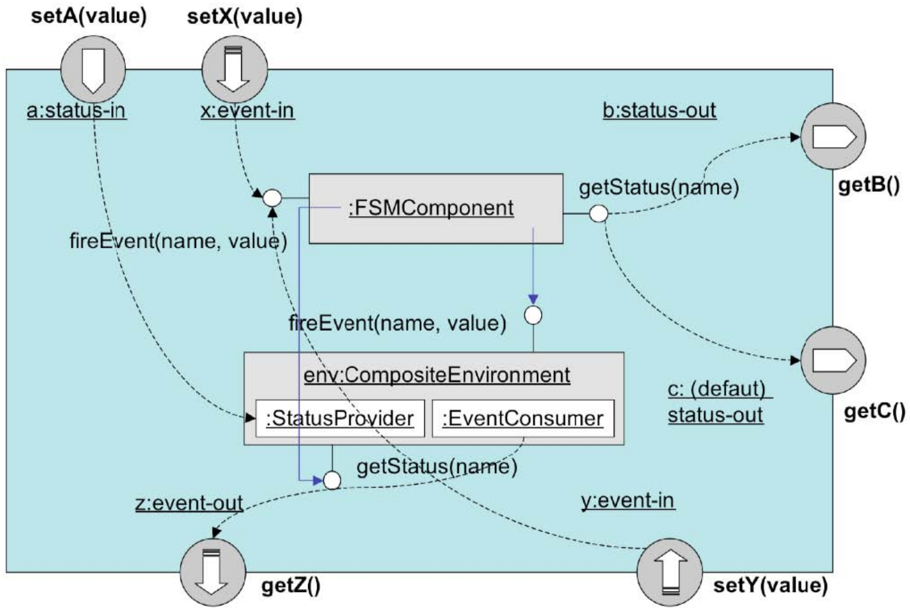

Fig. 10.

Unfortunately, in order to map Status–Event components into a Java Bean we have to effectively lose most of the distinction between Status and Event at the Bean level, both are properties; the differences between the properties are purely in the internal 'wiring' and in the expected way in which those properties will be accessed externally. While these wrapped Beans still provide explicit documentation of the status/event distinctions and also, to some extent, a more natural way of specifying the various status–event relations, it does lose the opportunities for more global reasoning and optimisation. In particular, we cannot throttle unneeded raw events or status-change events. Happily, the entire system formed by binding components with a configuration then forms a new component. So this compound component can also be wrapped into a Java Bean meaning that internally it can make use of the full richness of the SE environment including throttling.

9 Summary

We have seen how XSED allows descriptions of systems that include both status and event phenomena to be included naturally and without having to prematurely transform the status into discrete events. The notation separates components, configuration, binding and annotation allowing reuse and flexibility, but also allowing global analysis (by hand as now, or in future automated) to feed into optimisation of the execution over the infrastructure. We also saw how the symmetric treatment of input and output allowed the external environment of configurations of components to be treated as a component alongside others. The transformation onto Java has created efficient implementations of XSED components and systems and Bean wrappers allow these to be embedded within existing infrastructure, notably ECT.

Note that at a low level XSED specifications are still implemented as discrete events – this is the nature of computers. The crucial thing is that the specifications themselves do not assume any particular discretisation of status phenomena into lower-level system events. For the analyst/designer this means they describe what they wish to be true, not how to implement it. At a system level this means appropriate mappings onto discrete events can be made based on analysis not accident. The difference between XSED and more event-based notations is thus rather like between arrays and pointers in C-style languages, or even between high-level programming languages and assembler.

Future work on the underlying notation includes: refinement to allow status-change events (such as when temp > 100°C); alternative basic component specifications (e.g. process algebra based); ways of naming components (e.g. URIs) to allow binding to be controlled through XML files and additional annotations. In addition we plan more extensive case studies including distributed examples where the efficiency advantages can be fully appreciated.

References

1. Dix, A.: Status and events: static and dynamic properties of interactive systems. In: Proc. of the Eurographics Seminar: Formal Methods in Computer Graphics. Marina di Carrara, Italy (1991)
2. Dix, A., Abowd, G.: Modelling status and event behaviour of interactive systems. Software Engineering Journal 11(6), 334–346 (1996)
3. Greenhalgh, C., Izadi, S., Mathrick, J., Humble, J., Taylor, I.: A Toolkit to Support Rapid Construction of Ubicomp Environments. In: Proceedings of UbiSys 2004 - System Support for Ubiquitous Computing Workshop, Nottingham, UK (2004), http://ubisys.cs.uiuc.edu/2004/program.html
4. Flintham, M., Benford, S., Anastasi, R., Hemmings, T., Crabtree, A., Greenhalgh, C., Tandavanitj, N., Adams, M., Row-Farr, J.: Where on-line meets on the streets: experiences with mobile mixed reality games. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2003, Ft. Lauderdale, Florida, USA, April 05 - 10, 2003, pp. 569–576. ACM Press, New York (2003)
5. Dix, A., Abowd, G.: Delays and Temporal Incoherence Due to Mediated Status-Status Mappings (part of report on Workshop on Temporal Aspects of Usability, Glasgow, 1995). SIGCHI Bulletin 28(2), 47–49 (1996)
6. Wûthrich, C.: An analysis and model of 3D interaction methods and devices for virtual reality. In: Duke, D., Puerta, A. (eds.) DSV-IS 1999, pp. 18–29. Springer, Heidelberg (1999)
7. Massink, M., Duke, D., Smith, S.: Towards hybrid interface specification for virtual environments. In: Duke, D., Puerta, A. (eds.) DSV-IS 1999, pp. 30–51. Springer, Heidelberg (1999)
8. Willans, J.S., Harrison, M.D.: Verifying the behaviour of virtual environment world objects. In: Palanque, P., Savernó, F. (eds.) DSV-IS 2000. LNCS, vol. 1946, pp. 65–77. Springer, Heidelberg (2001)
9. Grossman, R., et al. (eds.): HS 1991 and HS 1992. LNCS, vol. 736. Springer, Heidelberg (1993)
10. Dix, A.: Finding Out -event discovery using status-event analysis. In: Formal Aspects of Human Computer Interaction – FAHCI 1998, Sheffield (1998)

11. Dix, A., Beale, R., Wood, A.: Architectures to make Simple Visualisations using Simple Systems. In: Proc. of Advanced Visual Interfaces - AVI 2000, pp. 51–60. ACM Press, New York (2000)
12. Goodger, B., Hickson, I., Hyatt, D., Waterson, C.: XML User Interface Language (XUL) 1.0. Mozilla.org. (2001) (December 2006), http://www.mozilla.org/projects/xul/xul.html
13. Nicol, G.: XTND - XML Transition Network Definition. W3C Note (November 21, 2000), http://www.w3.org/TR/xtnd/
14. Abowd, G., Allen, R., Garlan, D.: Using style to understand descriptions of software architecture. In: Notkin, D. (ed.) Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1993, Los Angeles, California, United States, December 08 - 10, 1993, pp. 9–20. ACM Press, New York (1993)
15. Ramduny, D., Dix, A.: Impedance Matching: When You Need to Know What. In: Faulkner, X., Finlay, J., Détienne, F. (eds.) HCI 2002, pp. 121–137. Springer, Heidelberg (2002)
16. Bacon, J., Moody, K., Bates, J., Chaoying, M., McNeil, A., Seidel, O., Spiteri, M.: Generic support for distributed applications. IEEE Computer 33(3), 68–76 (2000)
17. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. In: Proc. 18th Annual ACM Symposium on Principles of Distributed Computing (PODC 1999), pp. 53–61. ACM Press, New York (1999)
18. Gelernter, D.: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems 7(1), 255–263 (1985)
19. Picco, G.P., Murphy, A.L., Roman, G.: LIME: Linda meets mobility. In: Proceedings of the 21st international Conference on Software Engineering, Los Angeles, California, United States, May 16 - 22, 1999, pp. 368–377. IEEE Computer Society Press, Los Alamitos (1999)
20. Wade, S.P.W.: An Investigation into the use of the Tuple Space Paradigm in Mobile Computing Environments, PhD Thesis, Lancaster University (1999)
21. Ponnekanti, S.R., Johanson, B., Kiciman, E., Fox, A.: Portability, extensibility and robustness in iROS. In: 1*st* IEEE Pervasive Computing and Communications Conference (PerCom 2003), 23-26 March 2003, pp. 11–19 (2003)
22. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. In: Centrum voor Wiskunde en Informatica. Advances in Computers, vol. 46. CWI Press (1998)
23. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. 14(3), 329–366