Structure of Event Files

Introduction

Event files (extension .events) contain a sequence of events as a result of client-server interactions where servers are usually components that implement ComMA interfaces. Events are commands, signals, notifications and replies to commands. They are defined in a ComMA interface signature.

Event files are not intended to be manually constructed by the users. Usually they are automatically generated from existing system logs and observed client-server communication. The file structure is optimized for machine processing.

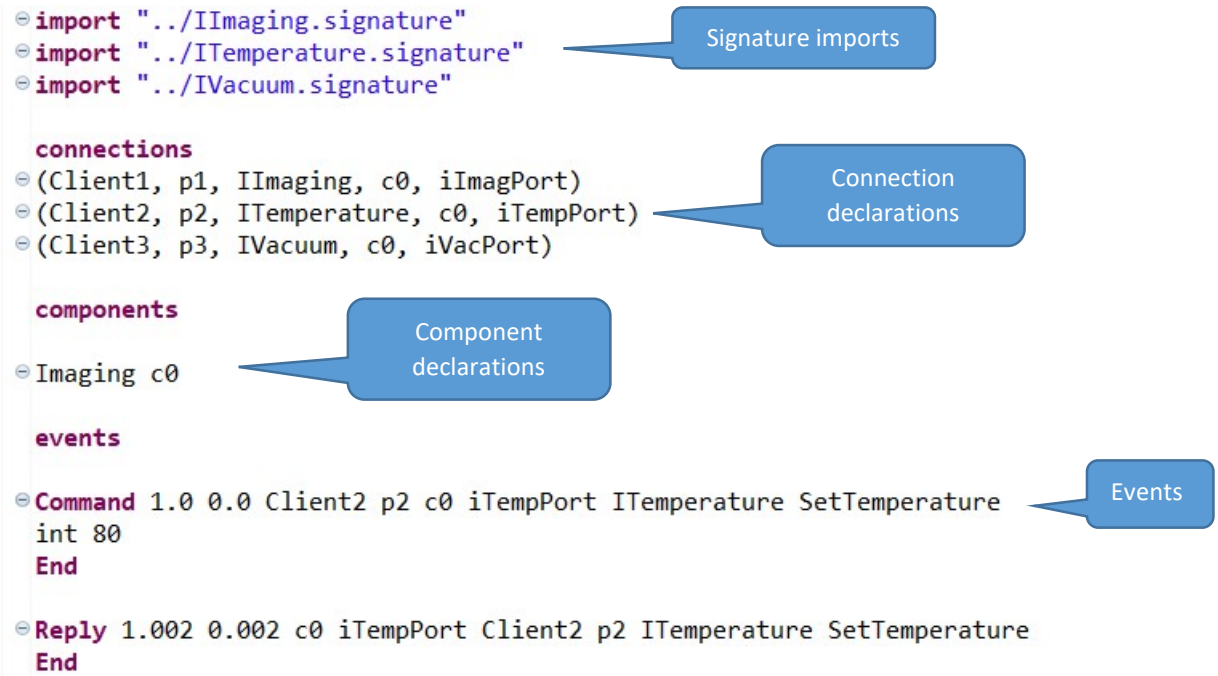
This document describes the format of event files. The target audience is engineers that implement event file generators and users that specify small trace files for experimentation purposes.

Structure

Event files have the following sections:

- Imports of the used ComMA signatures (optional)
- Declaration of connections
- Optional declaration of component instances
- Events

Important: event files must not contain comments.



Connections and Component Instances

A connection represents the usage of an interface by a single client. The interface is provided by a server.

Connection structure:

(client_id, client_port, interface_name, server_id, server_port)

Client and server identifiers are symbolic names that denote communicating parties. The identifiers are unique in the scope of the events file. A communication party can play the role of a client in one connection and the role of a server in another one. For example, a component instance with a given name (declared in the component declarations section) can be a server for connections on the provided ports, and a client for the connections originating from its required ports.

An events file has to contain at least one connection declaration and may optionally contain declarations of component instances. If component instances are declared and used in the connections then the port names and their interfaces as given in the component definition must be used consistently in the connections. This is explained below.

In the example, a single component instance *c0* of type *Imaging* is declared. The definition of *Imaging* is as follows (as a part of a definition not fully shown here):

```
component Imaging

provided port IImaging iImagPort
provided port ITemperature iTempPort
provided port IVacuum iVacPort
```

The names of the ports in the definition have to be used in the connections in which c0 participates. In the example file there is one client for each port (*Client1*, *Client2*, and *Client3* respectively). This is captured in the declaration of three connections. As can be seen, the server ports and the associated interfaces match the component definition. The client identifiers and the client ports in this case are just names, there is no component declaration for them.

It is possible that a component instance is a client for another component instance. In this case the client port name in a connection must be equal to the name of a required port in the component definition. Circular connections in which a communicating party is a direct client of itself in the same connection are not allowed.

It is allowed to declare multiple component instances. Multiple component types can be used in component instance definitions.

Remark: currently there is no validity check for a proper match between the port names used in connections and events, and the port names in component definitions.

Events

Events are occurrences of commands and their replies, signals and notifications as defined in a ComMA signature.

An event has the following structure:

EventId
Event_description
…
parameter
parameter
…
End

EventId is optional. If given, it is placed on the line before the event. It is useful for visualization purposes and for facilitating the correspondence between the error reports and the events where the error is detected.

The Event_description gives the type and the name of the event and time information about its occurrence. An event may have zero or more parameters, each on a separate line. The keyword 'End' indicates the end of the event description, it must always be present and placed on a new line.

The event description has the following format:

event_type timestamp time_delta source_id source_port target_id target_port interface_name event_name

Event descriptions are placed on a separate line.

- *event_type*: one of Command, Signal, Notification, Reply
- *timestamp*: a representation of the time of the event occurrence. Can be either an epoch time given as a real number or a date-time format: YYYY-MM-DD-HH:MM:SS.MSEC
- *time_delta* is the time difference between two consecutive events. It is measured in seconds and given as a real number. The first event has time_delta 0.0
- *source_id, source_port, target_id, target_port* must match the connection declaration. In case of commands and signals, source_id and source_port are the connection's client_id and client_port; target_id and target_port are the connection's server_id and server_port. In case of replies and notifications, the situation is reversed
- *interface_name* and *event_name:* the name of the event's interface and the event name. In case of a reply, the name is the name of the command

Parameters

An event may have zero or more parameters. Commands may have **in**, **inout**, and **out** parameters. A command event gives its **in** and **inout** parameters in the order of their definition in the signature.

A reply to a command gives parameters in the following order: **inout** and **out** parameters (if any) in the order of their definition, reply value (if any)

Every parameter is given in a separate line and has the following structure:

type_indicator value

Type Indicators

- Simple types: int, bool, real, string, bulkdata
- Enum types: enum
- Record types: record
- Collection types: vector

Simple and Enum Values

- Simple types:
    - int: 1, 123
    - bool: true, false
    - real: 123.0, NaN
    - string: "some string"
    - bulkdata: the value of a bulkdata is just an integer that indicates the number of bytes in the data

- Enum types: the name of the type and the literal, e.g. `Status OK`

Some examples:

```
Command 1.0 0.0 Client2 p2 c0 iTempPort ITemperature SetTemperature
int 80
End

Reply 2.005 0.005 c0 iImagPort Client1 p1 IImaging PrepareImage
enum Result Ok
End
```

Record Values

Record values have the following general structure:

Record_type_name field_values END

If the record type is defined in a ComMA interface, the name looks like:

_commaInterface interface_name type_name

If the record type is defined in a type file the name is just the name of the record type.

Field values are enumerated with a space as a delimiter and without giving the field names. Please note that type indicators for field values are not given either.

Examples:

Record type defined in an interface:

```
signature ITest

types

enum Status {OK Failed}

record Point {real x, real y}
```

Record value of type *Point*:

```
Reply 0.4 0.04 Server ev Client1 p ITest getOnePoint
record _commaInterface ITest Point 0.0 0.0  END
End
```

Type indicator

Record value

Nested record type defined in an interface:

```
record Point {real x, real y}

record Arrow {Point begin, Point end}
```

Record value of type Arrow expressed in ComMA expression syntax:

```
Arrow{begin = Point{x = 0.0, y = 0.0}, end = Point{x = 0.0, y = 0.0}}
```

The same record value as a parameter in events file:

```
Reply 0.4 0.04 Server ev Client1 p ITest getArrow
record _commaInterface ITest Arrow _commaInterface ITest Point 0.0 0.0  END _commaInterface ITest Point 0.0 0.0  END  END
End
```

Vector Values

Vector values have the following general structure:

Base_type_indicator size values END

Examples:

Vector type of records defined in an interface:

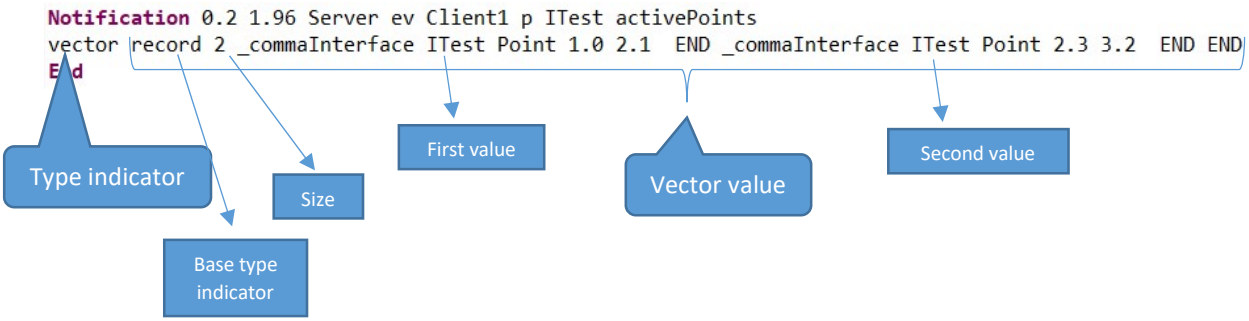
```
record Point {real x, real y}

vector Points = Point[]
```

Value of type *Points* expressed in ComMA expression syntax:

```
<Points>[Point{x = 1.0, y = 2.1}, Point{x = 2.3, y = 3.2}]
```

The same vector value as a parameter in events file:



Appendix

Grammar of parameters:

Parameter ::= TypeIndicator Value

TypeIndicator ::= ‘int’ | ‘bool’ | ‘real’ | ‘string’ | ‘bulkdata’ | ‘enum’ | ‘record’ | ‘vector’

Value ::= INT_VALUE |
BOOL_VALUE |
REAL_VALUE |
STRING_VALUE |
EnumValue |
RecordValue |
VectorValue

EnumValue ::= EnumTypeName EnumLiteral
EnumTypeName ::= ID
EnumLiteral ::= ID

RecordValue ::= (‘_commaInterface’ InterfaceName)? RecordTypeName Value+ ‘END’
InterfaceName ::= ID
RecordTypeName ::= ID

VectorValue ::= BaseTypeIndicator Size Value* ‘END’
BaseTypeIndicator ::= TypeIndicator
Size ::= NON_NEG_INT

The terminals INT_VALUE, BOOL_VALUE, REAL_VALUE, ID and NON_NEG_INT are defined as regular expressions:

INT_VALUE ::= (‘-’)? (‘0’..’9’)+

BOOL_VALUE ::= ‘true’ | ‘false’

REAL_VALUE ::= (INT_NUMBER '.' (‘0’..’9’)+ (('E' | 'e') INT_VALUE)?) | ‘NaN’

ID ::= ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*

NON_NEG_INT ::= (‘0’..’9’)+

STRING_VALUE denotes any sequence of characters surrounded by single or double quotes.