replay: Visualising the Structure and Behaviour of Interconnected Systems

Alex Murray **Duncan Grove**

Defence Science Technology Organisation
PO Box 1500, Edinburgh, South Australia 5111
Email: alex.murray@dsto.defence.gov.au

Abstract

Visualisation is often used to help understand complex systems and in particular scale-free networks which are present in many systems, from object-oriented software, to real-world and on-line social networks. While a number of tools already exist to visualise these systems, most focus on presenting the network as a whole and neglect to include information on the possibly concurrent behaviour of individual nodes. In this paper we present *replay* which aims to meet these demands, by visualising both the structure and evolution of the network through time, as well as the behaviour of individual nodes and the communications between nodes. We describe the unique and novel aspects of *replay*, including its three different but related visualisations of the underlying system, as well as its plug-in architecture, which allows *replay* to be extended and applied to visualise different networked systems. We also demonstrate the utility and flexibility of *replay* with a number of real-world visualisation examples, as well as present possible directions for future work.

Keywords: Graph and network visualisation, concurrency visualisation, interaction visualisation, interconnected systems, concurrent systems.

1 Introduction

Visualisation is increasingly being used to aid understanding of complex systems. In particular, scale-free networks [5] have recently become a focus for visualisation [26, 16], as a means to understand their underlying structures and hierarchies, as well as their evolution through time [19]. Many networked systems have been found to exhibit scale-free properties, ranging from social networks [5] (both real-world and on-line) to object-oriented software [30, 8, 24].

By their nature, scale-free networks are quite complex, comprising many nodes with numerous edges, and hence visualisations have focused on ways of simplifying the overall visualisation while still retaining the salient features of the network [16, 19]. These methods concentrate on visualising the network as a whole, and so while these tools have generally been successful in helping to understand the overall network structure, they provide little capacity for insight into the detailed behaviour of individual nodes – which we define as the node's activity and any messages it exchanges with other nodes.

In many cases, understanding the behaviour of individual nodes is crucial to properly understanding the overall behaviour of the network, since the evolution of the network through time is critically affected by the actions of its nodes. For example, the need for visualisations that show both the network's overal structure as well as the concurrent behaviour of its individual nodes has been identified as an educational tool to aid in the understanding of object-oriented software, especially the interactions between concurrent objects and how this influences the object graph as a whole [9].

We believe that to truly understand complex, networked, concurrent systems, visualisation tools must be capable of effectively exploring these systems at both macroscopic and microscopic levels of detail, while sweeping arbitrarily backwards and forwards through time.

We have therefore developed a visualisation tool called *replay* to meet these requirements. Section 2 describes the system while Section 3 presents some case-studies showing *replay* in action. We then describe related work in Section 4, future directions for our research in Section 5 and conclude in Section 6.

2 An overview of *replay*

replay was designed with a number of features for visualising complex, concurrent networked systems: a simple event based data model, three different but related visualisations of the underlying event model which are always synchronised, the ability to filter information, and a plug-in based extension system. Each of these features will be described in the following sections.

2.1 The *replay* event model

The primary elements represented within *replay* are *nodes*, *edges*, *activities* and *messages*, where nodes can be executing activities and are connected via edges to form the graph, and messages are sent between nodes along edges within the graph. *replay* employs a simple event-based data model which allows the behaviour and structure of diverse concurrent networked systems to be visualised. The plug-in interface (described in Section 2.7) provides programmatic access to drive the generation of events, allowing arbitrary systems to interface with *replay* at runtime using a diverse range of communication mechanisms. Each event specifies the time at which it occurred, as well as identifying the elements concerned. The four basic elements (*nodes*, *edges*, *activities* and *messages*) are all uniquely named within separate name-spaces

and can have arbitrary properties. The following describes the standard events associated with each element.

Node create / set properties Specifies the unique identifier for the node and a list of associated properties for the node in question (such as a label or colour).

Node delete Deletes the node with the given identifier.

Activity start Specifies the start of a uniquely named activity upon a particular node with a list of associated properties for the activity (such as colour or level).

Activity end Specifies the end of the activity with the given identifier.

Edge create Specifies the unique identifier for the edge, the identifiers for the head and tail nodes of the edge, whether the edge is unidirectional or bidirectional and a list of associated properties for the edge in question (such as a label, colour or weight etc).

Edge set properties Specifies the unique identifier of an edge and a list of associated properties to set for the edge at the given time of the event.

Edge delete Deletes the edge with the given identifier.

Message send Message send events specify an identifier for the message, the identifier for the sending node, a potentially associated edge via which the message travels and a list of properties for the message (such as a human readable description to display in the various visualisations). To model causality of message events, these events also specify a parent message which caused this message send to occur.

Message receive Specifies the unique identifier of the message and the node which is receiving the message.

Since all events specify a time-stamp, *replay* is able to reconstruct the sequence of events for the system through time, and allows the ability to step through the sequence both forwards and backwards through time. From the sequence of events, *replay* constructs three different but related views of the system. Figure 1 shows a screen capture of the main *replay* window displaying these three views:

Timeline view This is positioned at the top of the window and shows the state of nodes and their interactions through time.

Causal message tree view This view is placed at the left of the window and is designed to show the causal relationship between messages sent / received between nodes.

Network graph view This is situated on the right hand side of the main window, and is designed to show the graph of nodes within the system and how they are interconnected, along with their individual states, at a given point in time.

2.2 Timeline view

The timeline view presents a two dimensional view of the behaviour of nodes through time. Nodes are listed along the vertical axis, while time is plotted along the horisontal axis. A number of visual attributes are used to show the different states of nodes through time:

Node lifetime A thin line drawn in the node's base colour is drawn from the time of the node create event to the time of the corresponding node delete event.

Node activity The timeline view represents activity in a similar way to the network graph, using the activity level to determine the intensity of the activity colour. A thick coloured line is drawn in the current activity colour / level, and runs from the time of each activity change event to the next. An activity level of zero (the idle state) is indicated by the absence of this line.

Message flow Message send / receive event pairs are indicated by arrows drawn from the node which sent the message to the node which received the message.

Current time The timeline clearly indicates the current point in time using a thin line, with the region in the past shaded behind it.

This view is designed to show the concurrency and message passing characteristics of the system across time, and is similar to existing visualisations for parallel message passing systems [18, 29, 14]. By pairing together events, the timeline view is able to clearly show the duration of each interval, such that communication patterns, message passing latencies, and active / idle times are clearly visible.

The timeline view allows the user to zoom in and out, providing an infinite zoom resolution to allow the exact timing of events to be clearly represented and determined.

2.3 Causal message tree view

While *replay* allows events to be stepped through sequentially, the representations provided by the other two views give limited insight into the causal relationship of messages within the system. To address this, *replay* includes a third view, the causal message tree. Message send / receive events specify an identifier for the current message, as well as a potential parent message identifier which refers to the message event (if any) which caused the current one. This allows the message tree to be easily specified and constructed. Message send and receive events for the same message are aggregated into a single entry within the tree, as the causality of these events is directly linked (the receipt of a message is always the result of the corresponding send).

This view is situated at the lower left of the window, and lists the node which sent the message on the left, along with the message label on the right. Nodes are coloured with their corresponding base / activity colour, depending on their activity level at the time of the message send event.

This view is similar to the message-order view of Causeway [27], a message oriented postmortem debugger, and provides a visual representation of the causality for the current message event. This view is particularly suited to analysing interactions between

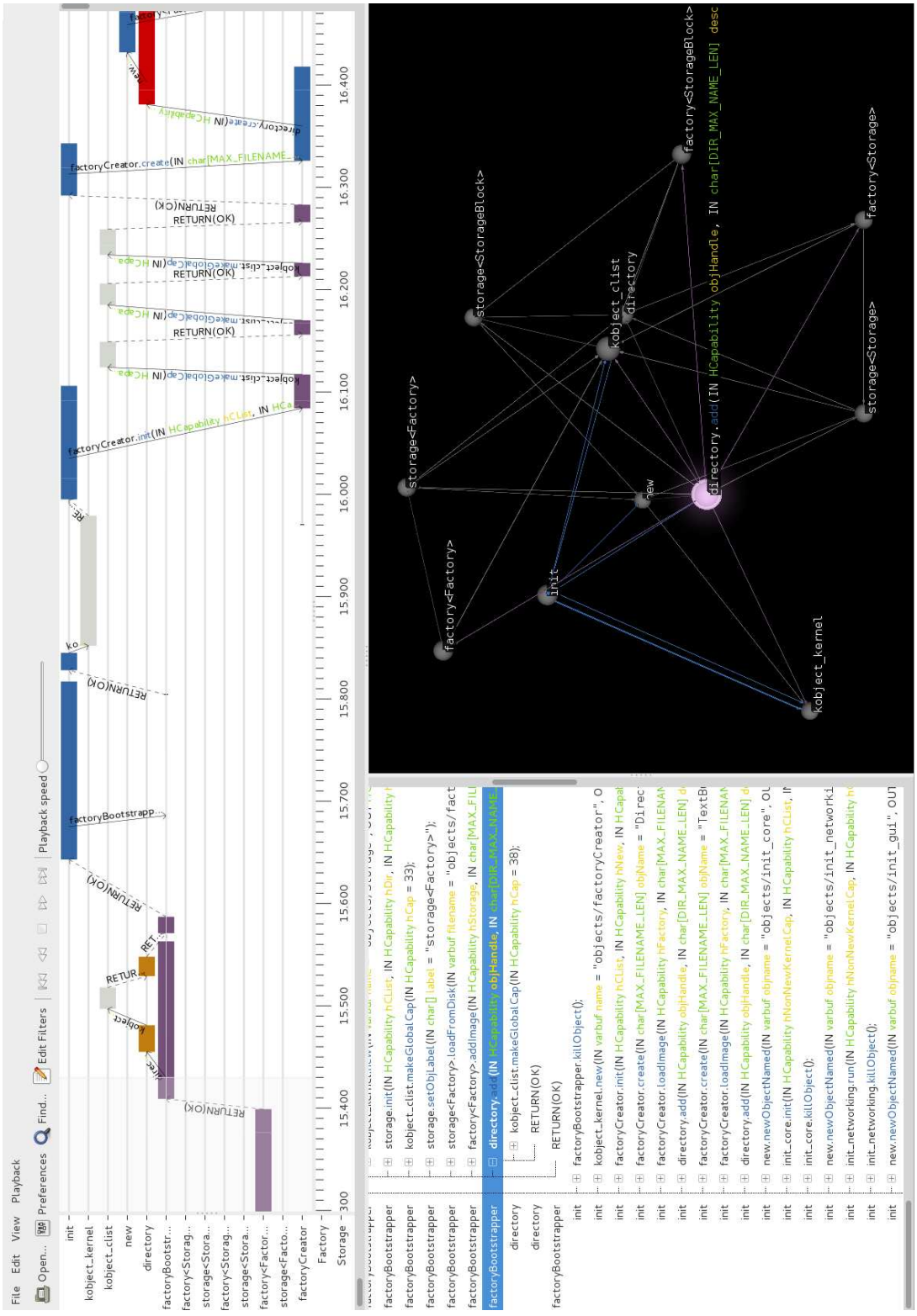

Figure 1: Main *replay* window showing the three unified views

nodes, such as in software development (i.e. representing the control flow via method calls between objects in an object-oriented software system).

2.4 Network graph view

The network graph view is situated to the right of the main window, and displays a three dimensional representation of the overall graph of the system. As the sequence of events is stepped through in time, the graph is constructed using the node create / delete and edge add / remove events. The primary purpose of this view is to show the connectivity of nodes within the system, as well as their current state, and finally to annotate the view with messages as they pass between nodes. As a result, a number of specific features have been incorporated into this view.

Like many existing visualisations which focus on the connectivity within a graph [16, 19] a force-directed model is used to layout the graph. All nodes repel each other with an inverse gravitational (Coulombic repulsion) force, while neighbors attract one another using a spring-modeled force. Unlike the two visualisation models cited previously, nodes in *replay* have a 'mass' which is proportional to the total number of connections they have. Nodes are then drawn with a size proportional to this mass (assuming a constant density), and the inverse gravitational repulsive force calculations take this value of mass into account. By adding this property, nodes which are highly connected (and hence, for example, have greater 'authority' as defined by [17]) are significantly larger and are placed further away from less connected nodes. This creates a visual representation where the highly connected nodes are easily discerned due to their placement and size within the overall graph.

Nodes are coloured using the 'color' property value, and activities are drawn as a glow around the node in the designated colour and at the designated intensity (using the 'color' and 'level' properties of the activity respectively). Multiple concurrent activities on a node result in blending of their respective colours at their respective intensities.

The graph is also annotated with the labels of message send and receive events as these events occur, along with the name for the corresponding node. By overlaying these labels alongside the node with which they are associated, the flow of messages within the system is able to be represented along with the state of the system as these events occur.

This view is able to be controlled by the user, providing the ability to center the view on a particular node of interest, zoom in and out, or arbitrarily rotate the viewpoint around the current center. By default node and edge properties are hidden, but are exposed when the user places the mouse over the node or edge of interest. This allows the graph to be easily understood by showing only the vital information, but provides an effective way to allow the user to reveal relevant information as required. Finally, the user can also interact directly with the graph and move the nodes within it to determine how this affects the overall graph layout.

2.5 Synchronisation of views and interaction

While each of the three views provides its own unique representation of underlying the system, we believe the real advantage of *replay* is the combination of all three views. As a result, all three views use similar representation (such as node colour) and remain synchronised at all times, to ensure a consistent representation of the event sequence, and hence the underlying system itself. This is an important feature, since it helps to highlight relationships between the views and allows the different information presented within the views to complement one another [28]. Also, by using consistent representations within all three views, *replay* reduces the cognitive load on the user to understand the underlying data. As a result, this frees the user, allowing them to process large amounts of complex data quite easily, due to the natural cognitive abilities of the human visual system [7].

replay also employs user interaction within all views to allow the user to jump to certain events, and to manipulate the displays of the views. For example, selecting a message within the message tree causes both the network graph and timeline views to jump to that event in the event sequence, and similarly, events can also be selected in the timeline view.

Finally, both the message tree and graph view allow the user to search for a message or node by name respectively to easily locate items of interest.

2.6 Filtering

In many large systems the number of nodes, and their interconnections and communications, can produce quite complex visualisations where the finer details of the system are obscured. To deal with this complexity, a number of techniques for automatically simplifying the overall graph structure have been explored [8, 19]. In *replay* we also provide a means for filtering the graph, providing the user with direct control over which properties to filter from the display as well as providing the ability to implement automatic filtering through the plug-in extension system as described in Section 2.7.

Filters can be created which specify a list of specific nodes, or a glob [12] style pattern to match the names of nodes, against which the filter is applied. The filter can then specify that these nodes are either grouped, or hidden, or can override the properties (colour etc.) of the nodes. Groups are then represented as the aggregate of their component nodes within the different visualisations. The timeline view uses a single entry which draws the timelines of the component nodes overlayed upon one-another, while the network graph represents a group as a single node with the combined mass of its components, hiding all internal edges between nodes within the group. Hidden nodes are removed from all views (and any edges or messages in which they are involved).

This allows the user to selectively hide extraneous information while retaining that which is pertinent to the current analysis. This in turn allows the user to reduce their cognitive load and hence focus on the problem at hand. The use of such filtering has been successfully demonstrated in the analysis of Annex object capability based software [23], which will be explored in Section 3.1.

2.7 Plug-in / Extension system

Originally *replay* was built as a tool to help analyse and debug the Annex object capability system and, as a result, was initially tailored to suit the specifics of Annex. However, it was soon realised that the different visualisations within *replay* could be very useful in analysing other systems including other object-capability / object-oriented programming systems or social networks. A plug-in system was developed to enable *replay* to be easily extended and used to visualise other diverse, inter-connected systems.

Plug-ins can be used to extend *replay* in multiple ways:

Event Sources The initial motivation for the development of the plug-in system was to provide support for the translation of custom data sets into the specific events described in Section 2.1. Hence plug-ins can provide and register event sources for the main *replay* application, allowing *replay* to easily support a wide range of systems. Multiple types of event sources are supported, including disk-based file sources for offline visualisation, or network connected sources for visualisation of live systems.

Analysis Plug-ins also have access to the various data-structures within the core application, such as the sequence of events, the node-edge graph, and the list of filters. This allows for a number of extensions to be implemented, such as performance analysis or automatic filtering by the creation of custom filters. As an example, a plug-in could easily analyse the graph at a given point in time to determine disjoint sub-graphs. By accessing the filter list, it could then create filters to select the nodes in each separate graph and override their colours. This would then provide a simple visual cue of the separate graphs to the user without the need for manual intervention.

Extended Functionality The plug-in system has been used to implement a number of the core features for *replay*, including playback controls (allowing the user to automatically play forwards and backwards through the events), as well as the filtering system presented in Section 2.6.

A number of plug-ins have been developed to extend the utility of *replay*.

Annex The original Annex specific code from *replay* was re-factored into a single plug-in which interprets the custom Annex event log and produces appropriate *replay* events. An example of the output from this is seen in Figure 1.

Java To demonstrate the utility of *replay* as a general tool for the visualisation and analysis of object-oriented programming languages, a plug-in is being developed to interface with the output from the OKTECH Profiler [2] for the Java Virtual Machine to allow generic Java programs to be visualised. This currently provides support to visualise the object reference graph through time, differentiating references obtained through object creation, method invocation and return value by using different colours for each. This plug-in also provides the ability to view the execution of methods upon objects through time including their method signatures.

Due to limitations in the OKTECH profiler and the nature of the Java garbage collector there is currently no support for determining when references are dropped, and so references simply accumulate in the graph. Even without this complete support, we believe that with this existing plug-in *replay* provides almost complete support for the visualisation of Java programs, for which a clear need has previously been identified [9].

FDR A plug-in has been developed to aid in the task of formal analysis of object-capability security patterns [22] which translates the output of the FDR [10] model checker into appropriate *replay* events. This will be discussed further in Section 3.2.

Graphvis A plug-in has also been developed to translate the Graphvis [11] dot-format graph descriptions into *replay* node and edge events to allow these graphs to be visualised in an interactive, three-dimensional display using the force-directed layout of the network graph view.

Causeway The previously mentioned Causeway message-oriented debugger was designed to debug concurrent message passing systems such as the object-capability E programming language [20] and the Waterken web server [4]. We have developed a plug-in to translate the Causeway message log format [1] into *replay* events to allow these systems to be visualised.

By separating the logic required to parse and interpret custom data sets into different plug-ins, we have been able to focus on the core visualisation technologies within *replay* itself. The following section will discuss the use of *replay* in the analysis of real-world systems, describing its utility and benefits as a general purpose visualisation tool.

3 Case studies

3.1 The Annex Capability System

The original motivation behind the development of *replay* was to develop a tool for debugging and analysing the security properties of the Annex object-capability system (which will be referred to as simply Annex for the sake of brevity). Annex serves as the Trusted Computing Base (TCB) in a number of secure devices developed by DSTO Australia [13, 23]. The Annex TCB is used to control the security policy for these devices, and hence the correctness of the Annex system is crucial to ensuring the security of the devices as a whole.

Within Annex, and other object-capability systems, objects can only communicate with one-another by message passing, and they can only pass messages if they have an appropriate capability which designates the other object. As a result the collection of capabilities which an object possesses defines the authority of the object within the system [20]. As capabilities can be delegated from one object to any other that they are already in communication with, the object graph (where objects are nodes, connected via capabilities) is a dynamic entity which is constantly changing as the system evolves. The ability to easily visualise this graph and hence quickly 'see' the security policy / posture of the system embodied by the graph was the primary motivation in the development of *replay*. Once the graph view was developed, it was also realised that as well as visualising the security properties of the system, the ability to visualise the behaviour of the system through time would also help in analysis and debugging. Hence the timeline view was added. Similarly, the need to track the causality of messages (what caused this message to be sent) was identified, and resulted in the addition of the message tree view.

replay has served as a significant tool in the development of Annex by allowing the entire execution of the system to be visualised. This has been particularly useful in a number of situations, some of which are summarised in the follow sections.

3.1.1 Timing and race-conditions

Annex objects interact by method calls / returns in a turn-based fashion using an asynchronous

promise [20] model. An object is active and uninterruptable while processing a call (and this defines a single turn for the object), but it is idle while waiting for the response to other calls it makes. As a result, while waiting on the result from a call an object has made, it can be invoked by either another incoming call, or a response to a different call it has previously made, which will start the execution of a new, independent turn. This feature of the Annex systems allows high levels of concurrency, since multiple turns (and hence multiple method calls) can potentially be interleaved without needing to wait for each to synchronously execute, and also allows a high level of parallelism to be achieved, while still guaranteeing atomicity between turns. However, without careful attention to turn boundaries this feature can also lead to potential race-conditions and security critical bugs. *replay* has proved useful in helping to track down these particular issues by clearly showing the concurrent and interleaved execution of calls within a single object. The Annex plug-in colours each different call separately and so the interleaving of different calls to a single object is clearly visible within the timeline view, as seen in Figure 2.

This Figure shows the execution of the *tortureAsync* application which comprises 1 driver object (*tortureAsync*) and 16 worker objects *oTortureAsync* repeatedly calling one-another, and is designed to stress-test Annex's message-passing performance. The execution of the top-most *oTortureAsync* object clearly demonstrates this interleaving. This object is initially called by the driver *tortureAsync* and starts execution (shown by the blue activity line) and proceeds to call 4 of the other worker objects, including itself. It then suspends execution to wait for the returns from these calls. Almost immediately a return is received from the first object which it called (again shown by the same colour blue activity line), at which point the initial call is resumed to store this result, and execution is again suspended. However, since the object is now idle, the call which it made to itself is now delivered, shown by the green activity line. This starts a new turn, which is separate from the one used to execute the original call (indicated by the different colours) and clearly demonstrates that these two calls have been interleaved on this object. In interleaving these calls, if the second call happens to modify state which the first call is expecting to remain constant, then a race-condition will result. However, the timeline view of *replay* clearly allows this to be identified and flagged to the programmer. It should be noted that the causal view will not help identify this same potential for error since it only highlights the causal, i.e. partial ordering, not total ordering.

Figure 2 also clearly shows the ability to measure the time taken to execute particular calls - the time taken to execute the call from *tortureAsync* to each of the worker *oTortureAsync*'s is clearly longer than the time taken to execute the calls made between each worker object.

3.1.2 Authority analysis

As the purpose of the *tortureAsync* application is to simply make repeated calls between each of the worker objects, there is no need for these objects to have capabilities to any other object within the system, except for the other worker objects. This design follows the principle of least authority, which states that an object should only have the minimum authority required to perform its intended function, and no more [21]. We can easily analyse the authority of the application by inspecting the object graph within the network graph view of *replay*, as shown in Figure 3.

From simple inspection of the Figure, we can see the 16 worker objects of the application situated in the bottom right, with the rest of the objects comprising the other applications of the system in the left of the Figure. It is clear that these are two distinct and separate graphs, i.e. there are no capabilities connecting the worker objects to the rest of the system. *replay* therefore provides the ability to quickly verify the intended security-related isolation properties of this system by simple inspection of the graph. While this is clearly useful, the usability of visual inspection decreases with the complexity of the system at hand. Therefore for more complex analysis, as previously mentioned, the plug-in system provides the ability to directly access various data structures maintained by *replay* (such as the graph structure), allowing programmatic analysis of various properties of the system to be implemented as needed.

3.2 FDR Model Checker

The FDR plug-in was developed to visualise the trace output from the FDR model checker [10] when analysing CSP [15, 25] models of object-capability security patterns [22]. Communicating Sequential Processes (CSP) is a process algebra used to describe concurrent message-passing systems and allows formal models of such systems to be constructed. The correctness of such formal models can be stated and tested using *refinement checks* which can be evaluated by the Failures-Divergences-Refinements (FDR) model-checker to ensure correctness of the system. Murray [22] describes the use of CSP to model object-capability security patterns and the use of FDR to test the security properties of such models: CSP is used to construct a model for the object-capability pattern, which expresses the desired security properties for the system as well as the potential behaviour of its components. FDR is then used to test whether these security properties hold, and if not will return a counter example of the system's behaviour which violates the properties. One such counter-example, taken from the work of Murray [22] in analysing the *Sealer-Unsealer* pattern for object-capability systems is as follows:

```
TheDriver.Alice.Call.null,
Alice.TheUnsealer.Call.Alice,
TheUnsealer.TheSlot.Call.null,
TheSlot.TheUnsealer.Return.null,
TheUnsealer.Alice.Call.null,
Alice.TheDriver.Return.null,
TheDriver.Bob.Call.null,
Bob.TheBox.Call.null,
TheBox.TheSlot.Call.TheCash,
TheSlot.TheBox.Return.null,
TheBox.Bob.Return.null,
Bob.TheDriver.Return.TheBox,
TheDriver.Alice.Call.null,
Alice.TheUnsealer.Return.null,
TheUnsealer.TheSlot.Call.null,
TheSlot.TheUnsealer.Return.TheCash,
TheUnsealer.Alice.Return.TheCash,
Alice.TheCash.Call.TheDriver
```

From this trace output alone, and with no prior background information as to the example, it is almost impossible to determine the error which this counter-example expresses. However when visualised by *replay* (Figure 4), one aspect stands out as anomalous.

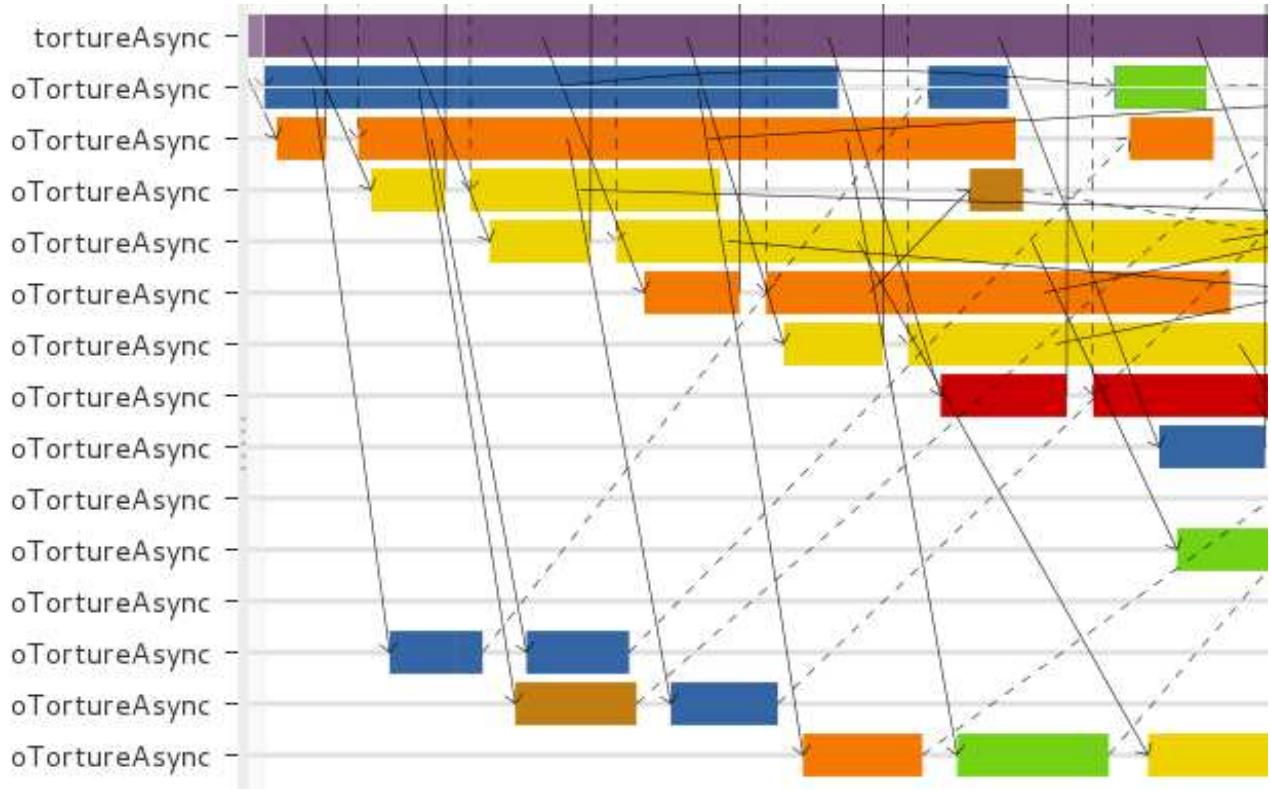

Figure 2: Timeline view of *replay* highlighting the interleaving of calls within objects in the *tortureAsync* application of the Annex system

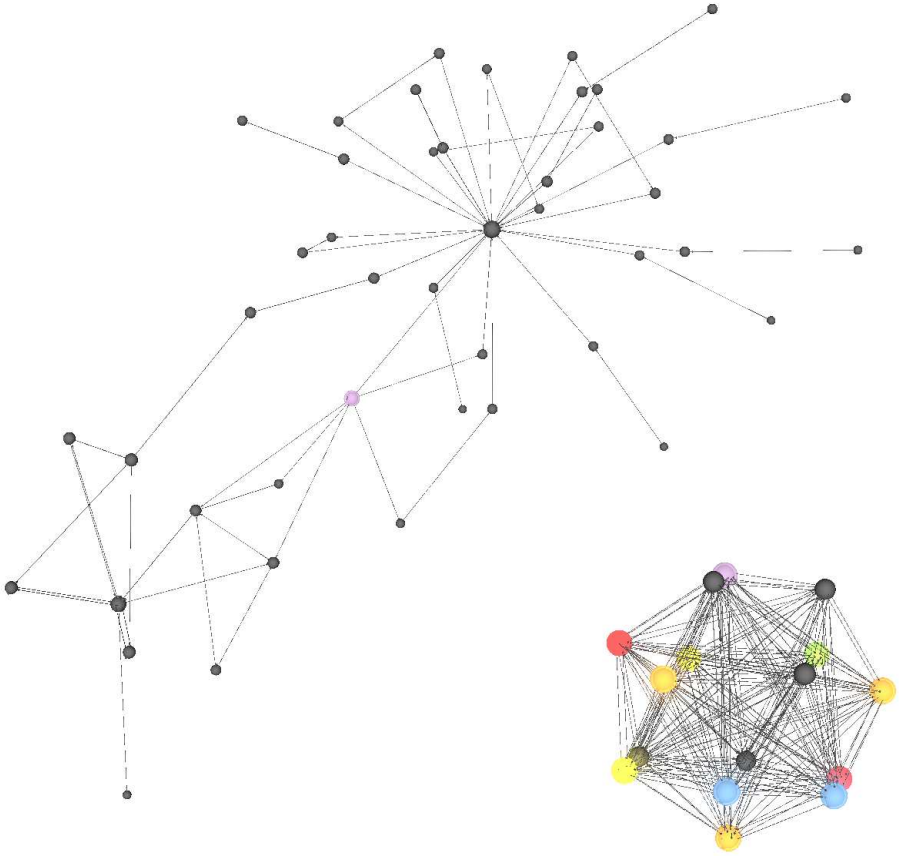

Figure 3: The Object-Capability Graph for the Annex system executing the *tortureAsync* application as depicted by the network graph view of *replay*. Note the complete isolation of the two graphs.

Figure 4: Message tree and timeline views of *replay* when analysing the output from the FDR model checker

Without any other knowledge of the system to inform our analysis, we can see the potential error clearly: the object *Alice* is called by *TheUnsealer* but then proceeds to return to *TheDriver*. This sequence of events is impossible in languages which impose strict call / return semantics, and indeed is responsible for the error in the model. By using *replay* to visualise this sequence of events we are able to quickly and easily identify the error in the model, a task which is not so easily accomplished by simply looking at the raw trace output alone. This example demonstrates the utility of *replay* in translating the user *unfriendly* output from this formal analysis tool into a much more easily understood visual form.

4 Related Work

Various *individual* aspects of *replay* are similar to existing visualisation tools. For example, the use of a three-dimensional force-directed network graph view is common for visualising scale-free networks [16, 19, 6, 3]; the causal message tree view is similar to that employed by Causeway [27]; and the time-line is also a standard technique for showing parallelism and message passing within systems [18, 29, 14] .

However, although the individual visualisations used by *replay* are not new, the *combination* of all three views plus an event model and plugin system make *replay* unique and interesting. We are not aware of any other tool which combines such disparate visualisations in such a coherent manner to provide a comprehensive system for understanding how networks change through time.

5 Future work

While Annex provided the initial motivation for the development of *replay*, the current and future directions for the project lay in applying the general information visualisation abilities of *replay* to a wider range of systems. The existing list of plug-ins already developed shows the ability of *replay* as a visualisation tool for general object-capability / object-oriented programming systems, and for general parallel, message passing systems.

Although *replay* has proven effective in visualising numerous software systems, we believe it would also be apt in visualising a wide range of existing real-world interconnected systems such as WWW hyperlink networks and computer networks including real-time data flows within such networks. *replay* could also be useful when applied within the field of forensic analysis of computer systems to visualise communication networks of suspects, and we believe *replay* would also be well suited to visualising real world social networks. The network graph view of *replay* is quite similar to existing social network graph visualisations [26, 16], and so is well suited to visualising the structure of such networks. We also believe the timeline view showing interactions through time, as well as the causal view showing the relationships between communications would provide valuable insight in understanding these networks which the previously cited tools do not provide. We also believe a similar approach could be used to visualise the transmission of email or instant messages, to determine the structure and behaviour of such communications.

While the manual filtering already provided by *replay* allows easy analysis by removing extraneous information, it does not yet apply to the causal message tree view, but this could be done in the future. It would also be useful to investigate the utility of applying automatic filtering and grouping mechanisms via the plug-in system, as well as implmenting various graph and performance analysis algorithms.

Finally, extensions to the plug-in system enabling other visualisations of the existing data structures, such as different layout algorithms for the network graph view, could also be developed.

6 Conclusion

In this paper we have presented *replay*, a novel tool for the visualisation of concurrent networked systems. We have shown the unique aspects of *replay* including its programmable event model and its three synchronised and related visualisations. The plug-in system, which allows *replay* to be applied to a wide variety of applications has also been presented and a number of existing uses of *replay* have been described, demonstrating its clear utility. Finally, we have contrasted *replay* against existing tools and presented possible future directions for this work. We believe that the use of a generic, programmable event model, the combination of the three different but consistent views of these events and an extensible plug-in system make *replay* a unique tool with both a high degree of usability and utility for the visualisation and analysis of networked, parallel systems.

References

[1] Debugging a Waterken application. Available online - http://waterken.sourceforge.net/debug/.

[2] OKTECH Profiler. Available online - http://code.google.com/p/oktech-profiler/.

[3] Walrus - graph visualization tool. Available online - http://www.caida.org/tools/visualization/walrus/.

[4] Waterken server documentation. Available online - http://waterken.sourceforge.net/.

[5] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286, 1999.

[6] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In *Proceedings of the International AAAI Conference on Weblogs and Social Media*, 2009.

[7] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[8] G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-laws in a large object-oriented software system. *IEEE Trans. Softw. Eng.*, 33(10):687–708, 2007.

[9] C. Exton and M. Kölling. Concurrency, objects and visualisation. In *ACSE '00: Proceedings of the Australasian conference on Computing education*, pages 109–115, New York, NY, USA, 2000. ACM.

[10] Formal Systems (Europe) Limited. *Failures Divergences Refinement - FDR2 User Manual*, 2009. Available online - http://www.fsel.com/documentation/fdr2/html/.

[11] E. R. Gansner and S. C. North. An open graph vizualisation system and its applications to software engineering. *Software - Practice and Experience*, 30:1203–1233, 1999.

[12] GNU. *GLOB(7) - glob - globbing pathnames*, August 2003. Linux Programmer's Manual 'man' page - http://www.kernel.org/doc/man-pages/.

[13] D. A. Grove, T. C. Murray, C. A. Owen, C. J. North, J. A. Jones, M. R. Beaumont, and B. D. Hopkins. An overview of the Annex system. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 341–352, December 2007.

[14] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.

[15] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[16] Y. Jia, J. Hoberock, M. Garland, and J. Hart. On the visualization of social and other scale-free networks. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1285–1292, 2008.

[17] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[18] E. Kraemer and J. T. Stasko. Creating an accurate portrayal of concurrent executions. *IEEE Concurrency*, 6(1):36–46, 1998.

[19] G. Kumar and M. Garland. Visual exploration of complex time-varying graphs. *IEEE transactions on visualization and computer graphics*, 12(5):805, 2006.

[20] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[21] M. S. Miller and J. S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *Proceedings of the 8th Asian Computing Science Conference (ASIAN03)*, pages 224–242, December 2003.

[22] T. Murray. Analysing object-capability security. In *Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08)*, 2008.

[23] C. Owen, D. Grove, T. Newby, A. Murray, C. North, and M. Pope. PRISM: Program Replication and Integration for Seamless MILS. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 281–296, Washington, DC, USA, 2011. IEEE Computer Society.

[24] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in OO programs. *Commun. ACM*, 48(5):99–103, 2005.

[25] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.

[26] L. Shi, N. Cao, S. Liu, W. Qian, L. Tan, G. Wang, J. Sun, and C. Lin. HiMap: Adaptive visualization of large-scale online social networks. In *Proceedings of the 2009 IEEE Pacific Visualization Symposium-Volume 00*, pages 41–48. IEEE Computer Society, 2009.

[27] T. Stanley, T. Close, and M. S. Miller. Causeway: A message-oriented distributed debugger. Technical report, HP Laboratories, 2009.

[28] M. Q. Wang Baldonado, A. Woodruff, and A. Kuchinsky. Guidelines for using multiple views in information visualization. In *AVI '00: Proceedings of the working conference on Advanced visual interfaces*, pages 110–119, New York, NY, USA, 2000. ACM.

[29] J. Yan, S. Sarukkai, and P. Mehra. Performance measurement, visualization, and modeling of parallel and distributed programs using the aims toolkit. *Software Practice and Experience*, 25(8):429–461, 1995.

[30] Y. Yao, S. Huang, Z. ping Ren, and X. ming Liu. Scale-free property in large scale object-oriented software and its significance on software engineering. *Information and Computing Science, International Conference on*, 3:401–404, 2009.