Rapid GUI Development on Legacy Systems: A Runtime Model-Based Solution

Hui Song, Michael Gallagher, Siobhán Clarke
Lero: The Irish Software Engineering Research Centre
School of Computer Science and Statistics, Trinity College Dublin
College Green, Dublin 2, Ireland
{hui.song, siobhan.clarke}@scss.tcd.ie, gallam10@tcd.ie.

ABSTRACT

Graphical User Interface (GUI) is a common feature for modern software systems, while there are still many legacy systems that do not have GUIs, but only provide text and commands for user interaction. In this paper, we report our experiment on using runtime models to support the rapid, generation-based development of simple GUIs for such legacy systems. We construct runtime models for the target system as an intermediate representations of the underlying system state, and in this way wrap the low-level interaction mechanisms of the legacy systems. After that, we visualize the models with a graphical editor. Due to the causal connection between runtime models and the runtime system state, users can monitor and control the system state by reading and writing the models, and in this way, using the graphical model editor as the GUI of the system. Based on the existing framework for runtime model construction and model visualization, it is possible to achieve the rapid development process of such GUIs in the form of high-level specification and automated generation. We experiment with this idea by using two existing frameworks, Sm@rt and GMF, to develop a series of GUIs for an electricity simulation system named GridLAB-D. We also enhance the existing Sm@rt framework with cache mechanisms in order to suit GUIs.

Categories and Subject Descriptors

H.5.2 [**User Interfaces**]: Graphical User Interface (GUI); D.2.m [**Software Engineering**]: Miscellaneous—*Rapid prototyping*

General Terms

Design, Management, Experimentations

1. INTRODUCTION

Graphical User Interface (GUI) is an important feature of software systems, providing an intuitive way for users to understand the output or running status of the target system, and helping users effectively interacting with the system. However, there are still many legacy software systems that do not provide user interfaces, especially the ones that target technical users. Take our own story as an example. In our research lab, a number of people are doing research on smart electricity grids, and they use a popular electricity simulation tool named GridLAB-D to simulate the experimental residential or local grid networks. A typical residential system simulated by GridLAB-D could be a house constituted by meters, heating systems, water heaters, etc. GridLAB-D does not provide a GUI in the current version. The only way to get the current status of the simulated systems is to read the rolling text on the terminal about the update of element states. Similarly, when you want to change the state of any element at runtime, the only way is to launch a command in the form of a HTTP query. This text and command-based interaction is frustrating when there are a big number of elements simulated in the system. Therefore, a proper GUI for GridLAB-D is needed.

In general, GUIs are usually required to be highly flexible. On one hand, a system itself often evolves, and thus the GUI should adapt accordingly to support the new data or functions. On the other hand, for the same system, users from different perspectives may require different GUIs, presenting systems in different ways or supporting different functions. Here, GridLAB-D represents an extreme exemplar. As a simulation framework, GridLAB-D supports the simulation of a wide range of systems, from low-voltage residential appliances to high-voltage national electricity distribution networks. Different systems have different kinds of elements to visualize in the GUI, and the elements have different states and relations. For the same residential system simulated by GridLAB-D, if the users focus on the internal environment of the house, then the GUI should better represent the appliance elements in a virtual house; Alternatively, if the users focus on the electricity current between the appliances and their consumptions, then the GUI should represent the abstract topology and electricity lines between appliances.

Such flexibility means that the development of a GUI cannot be done only once, but has to be ready for customization and evolution. In other words, the development of GUIs should be a rapid or agile process: Developers could start from a small prototype, and incrementally add features or tune appearance based on discussion with end users.

In this paper, we report a rapid GUI development approach based on runtime model technology. The basic idea is to first represent the runtime system data that is useful for the GUI as a structural and dynamic model, and then use

MRT'12, October 02 2012, Innsbruck, Austria

the model-based visualization techniques to represent this runtime model. With the help of our runtime model construction framework Sm@rt [9, 8], and the Eclipse model visualization framework GMF[1], it is possible to achieve a rapid proces: Developers only need to provide three models to specify the types of the runtime data, the system management capabilities to retrieve and update these data, and the graphical representation of the data. From these specifications, the frameworks automatically generate a GUI in the form of an Eclipse plugin.

A main challenge to use runtime models for GUIs is how to efficiently maintain the causal connection between the system and the model. The GUI users require that all the displayed data are synchronized to the system state on time, and in the same time do not like the synchronization to cause much delay. This is hard to achieve because of the big scale of system data and the time-consuming API invocations to get and set the data. To meet this challenge, we propose and implement an on-demand cache-based causal connection maintenance mechanism for the runtime models.

The contribution of this paper can be summarized as follows. First, we discuss the role of runtime models in the development process of a specific type of software that needs to graphically represent the ever-changing state of the real world or the underlying systems, such as the GUIs. Specifically, a runtime model captures the intermediate data representation as a standard and dynamic model, and enable the traditional graphical modeling tools to be used as interactive GUIs for the underlying system. Second, we reveal one of the main challenges for runtime models in such usage, i.e., efficient synchronization between the system and the model, and provide a simple solution from the perspective of causal connection maintenance.

The rest of this paper is organized as follows. Section 2 describes the requirement of GridLAB-D GUI and its development process. Section 3 presents the construction of runtime models. Section 4 describes how to visualize the runtime models to form the GUI, and how to meet the challenge of causal connection for GUI purpose. Section 5 evaluates the approach on its usages and the development process of the final GUIs. Section 6 discusses the related approaches and Section 7 concludes the paper.

2. APPROACH OVERVIEW

In this section, we describe the GUI development example that will be used across this paper, and briefly summarize the development process for this GUI.

2.1 The GUI for GridLAB-D

GridLAB-D is an open source simulation framework for electricity systems. It simulates electricity-related elements, such as generator, transformer, appliance, etc., as C++ objects. Each object has a set of attributes, and their values describe the current state of the object. The simulation core manages these objects at runtime and controls their attribute values according to inner constraints in order to simulate the physical world. GridLAB-D provides a set of default classes to define the types of objects, their attributes, and the constraints of and between attributes. The users launch the simulation by instantiating objects from these classes, with initial attribute values.

[1] http://www.eclipse.org/modeling/gmp/

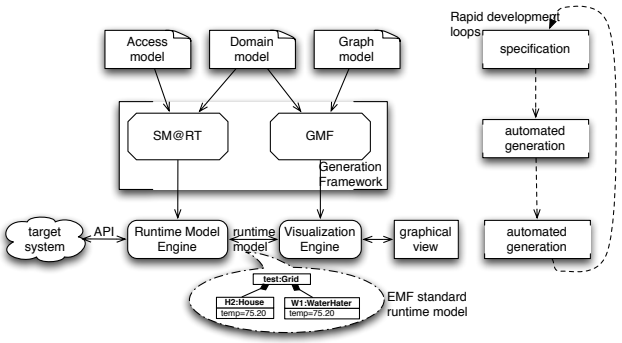

Figure 1: Rapid GUI development process

GridLAB-D provides a real-time mode for interactive simulation. It provides an HTTP-based API to launch reading and writing commands to all the attributes. But until the latest version (v2.2), GridLAB-D does not provide a GUI. All the interaction should be done through text outputs and manual commands. In order to test and represent the simulation in an intuitive way, our colleagues put forward the need of a GUI as follows. The GUI should represent the selected objects, their key attribute values, and the connections between them in a graphical view, intuitively reflecting the current simulation state. From the overall view, users should be able to select specific elements and see their inner structures or attributes in a separate view. The elements and the attribute values should be synchronized to the system, and users should be able to change them via the GUI.

Based on the basic requirement, there are also a number of detailed ones related to what objects or attributes should be selected, how to draw the specific class of objects, whether to display a given attribute on the main editor or the property view, how to deal with the object hierarchy (on the same editor, or nested editors), etc. However, these detailed requirements are different from user to user, depending on the scenarios they want to simulate. Based on the difference, and also on the difficulty to elicit the exact requirement from the users, a rapid development is most suitable.

2.2 The development process

Based on the runtime model technology, we adopt a rapid, generation-based approach for the GUI development. Figure 1 illustrates this development process.

The artifacts in this development process are three high-level specifications: The *domain model* specifies what types of system data will be visualized in the GUI, the *API model* describes how to retrieve and update these data, and the *graph model* defines how to visualize each type of data in the GUI. These three artifacts are in the forms of the Ecore meta-model, the access model defined in Sm@rt , and the graphical and mapping models defined by GMF, respectively. From these artifacts, the Sm@rt and the GMF generators automatically generate the runtime model engine and the visualization engine. The former maintains a runtime model in the form of a set of EMF elements conforming to the domain model. The runtime model has a causal connection with the state of the target system (in our example, the objects and their attribute values in the GridLAB-D simulator), and this causal connection is maintained by the engine by invoking the get and set methods in the system's man-

agement API. Based on this dynamic runtime model, the visualization engine maintains a graphical representation for each model element, and provides the auxiliary operations for reading and writing them.

The runtime model plays the role as an intermediate representation between the system data and the GUI. It enables the usage of a model-based visualization tool such as GMF in GUI development, from the following three aspects. First, it shields the low-level HTTP commands for the getting and setting of the system data, and organizes the data in a standard form as EMF models, which is accepted by GMF. Second, it organizes the raw data in software concepts (such as classes, interfaces, variable values, etc.) into the domain-specific, real world concepts (such as houses, applicants, meters, etc.), so that the content can be directly presented in the GUI. Third, the model is dynamically and bidirectionally synchronized with the system state, and thus the visualization of this model can be used for realtime system monitoring and reconfiguration

As illustrated by the right part of the figure, the development is constituted by infinite loops. Each loop starts from defining or revising the data types in the domain model. Following the revision on the domain model, the new API invocations are defined in the API model, and the graph model is changed to introduce new graphical strategy. From the new artifacts, the framework automatically generates the engines to support the GUI at runtime. In each loop, all the development effort is with abstract, model-level artifacts. After each loop, there is a runnable GUI for use, display, or discussion purpose.

There are two major issues related to such a rapid development process. The first issue is how to provide an abstract way to specify management API, and how to automatically generate the engines to synchronize the model and the system state. We utilize a general purpose runtime model construction framework named Sm@rt [9, 8] on this. The second issue is how to visualize the model and make the visualization suit the GUI purpose. Specifically, because of the huge data scale, the time-consuming data access, and the various frequency between the GUI updates and the system changes, it is hard to synchronize the system and runtime model both efficiently and in a timely manner. To address this, we introduce a new caching technique into the Sm@rt framework, specifically targeting the use of runtime models for GUI purposes. We present how we address these two issues in the following two sections, respectively.

3. CONSTRUCTING RUNTIME MODELS

In a previous paper, we introduced a meta-modeling framework for runtime models named Sm@rt (Supporting Model at Run-Time). The framework has three basic components, as shown in the left part of Figure 1. The *language* helps developers describe the types of system data and the access of system API for reading and writing the data. From the specification, the *generation engine* automatically generates the *runtime engine* that represents the system's runtime data as a set of dynamic model elements, and maintains the causal connection between the system and the model. A new version of this framework is implemented on Xtext[2], with a text-based modeling language for specifying API access, and a generation engine to the pure Java code.

[2] http://www.eclipse.org/Xtext/

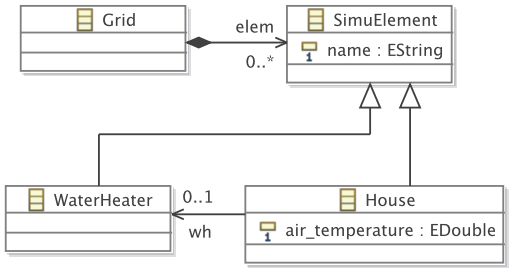

Figure 2: Meta-Model of the residential system simulated in GridLAB-D

```
1 invocation GetState{
2   operation get SimuElement->* {
3     feature.EType.name = 'Double'
4   }
5   invoke returning String {
6     val url = new URL("http://localhost:10001/" +
7       element.name + "/" + feature.name)
8     connection.setRequestMethod("GET");
9     val in = ... //get a BufferedReader
10    var s : String
11    while ((s = in.readLine()) != null)
12      if(s.contains('value'))
13        return value //skip string parsing
14  }
15  post (Double d) {current.eSet(feature,d) }
16 }
```

Figure 3: Access model of GridLAB-D API

Figures 2 and 3 illustrate the input specifications for GridLAB-D. Figure 2 is the meta-model that defines three basic data types we want to visualize in the GUI: a root class `Grid`, and two element classes `House` and `WaterHeater` for the simulated residential elements with the same names. Each simulated element class has a `name` attribute, and several other attributes to describe its simulation states. Figure 3 shows an excerpt of the access model, describing how to retrieve the attribute value through the web-based API provided by GridLAB-D. The `invocation` block defines a code snippet for the API access that has a specific function for data retrieval or update. The `operation` block (Lines 2-5) specifies the scope of this snippet, i.e., any attribute of the SimuElement class or its subclasses. The subsequent `invoke` block defines the concrete code snippet: We first create a URL with the element name and the attribute name, and then send an HTTP GET request to this URL to get the XML-format string. Finally, we parse the string to retrieve the attribute value. The return value of this snippet is of type `String`, which is converted to `Double` and used by the `post` block to update the attribute.

From these specifications, the Sm@rt framework automatically generates the runtime engine that maintains a set of instances of the classes defined in the meta-model. As standard EMF model elements, these instances form the runtime model of the simulated system under GridLAB-D. External model users read and write the model through standard EMF operations, such as get, set, insert, etc. The engine maintains the causal connection between the model and system through the generated aspect code that intercepts the model operations, and synchronizes the target at-

tributes. For example, if the user reads the attribute `air-temperature` on an `House` element, the engine will intercept this operation, and invoke the system according to the template code snippet defined in Figure 3, and return the value. The propagation is based on a set of synchronization strategies hard coded in the common runtime engine [9, 8].

4. CONSTRUCTING THE GUI

4.1 GMF and model visualization

We use the Eclipse Graphical Modeling Framework (GMF) to visualize the runtime model. GMF is a generative framework, which generates a graphical model editor in Eclipse (as shown in Figure 1), where each model element is represented by a shape (such as a rectangle, an ellipse, or a icon), and their relations are drawn as connection lines. The values of key attributes will be displayed together with the shapes, and other attributes are represented separately by a default property view. GMF also provides a set of auxiliary functions along with the visualization, such as the main and context manuals, sidebars, and automated layout, etc.

We provide two inputs for GMF to yield this GUI as above. The first input is the same meta-model in Figure 2, describing the types of elements in the runtime model. The second input is the graph model that defines how each type of model should be drawn in the graphical editor. In our example, we assign each class in the meta-model with an svg-format image file that indicates the intuitive meaning of the elements in the physical domain, and specify the key attributes to be drawn on the main editor.

From the two inputs, GMF generates the visualization engine to support the graphical representation of the model. The working principle of this engine can be roughly described as follows. When the main editor is opened, the engine first obtains the `elem` child list from the root `Grid` element, and draws the shape for each of these elements. For the attribute values that need to be drawn along with the shapes, the engines also obtains their values in this stage. After the initialization, the GMF runtime engine will automatically refresh the elements and the key attributes if the main editor is re-painted, and refresh the detailed attributes only if the element of these attributes is selected and the property view is active. To keep the GUI fresh, we also make the engine refresh the main editor and the property view regularly (every 1 to 10 seconds, depending on the requirement of different versions).

GMF was originally used to provide graphical editors for static models, and thus it is designed based on the assumption that the underlying models will not change unless users explicitly change them through the editor, and thus the reading of models is not time-consuming. However, the runtime models do not satisfy these assumptions. The system state is changing all the time, and thus if the runtime model is not synchronized with the system on time, the GUI user will not get the correct representation of the system. However, since the invocation of GridLAB-D's HTTP-based API is time consuming, and the runtime model usually contains a lot of elements and attributes, synchronizing the model and the system frequently and entirely is not tolerable. Considering the fact that the GUI usually just represents a small part of the system state at the same time, and the GMF-generated engine will not read the inactive part of the model, we choose an on-demand way to maintain the causal connection between the model and the system. That means the runtime model engine synchronizes the model and the system only when the visualization engine invokes the model reading methods, and it only synchronizes the parts mentioned by the invocations. Such on-demand synchronization is also the default mechanism by Sm@rt .

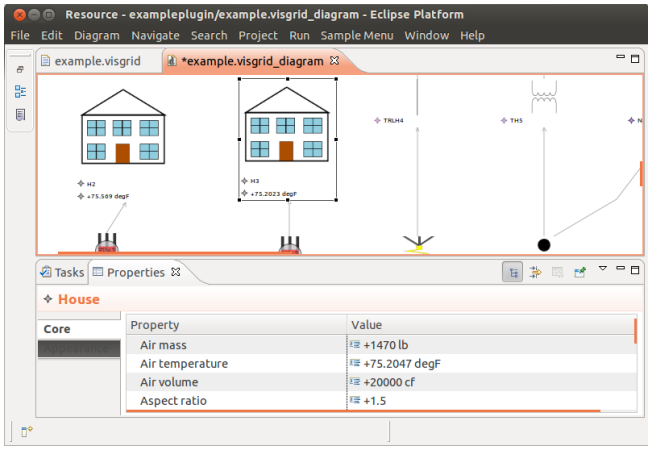

Figure 4: The screenshot of the final GUI

However, the purely on-demand synchronization mechanism means that every time the visualization engine retrieves a value from the model, the runtime model engines has to synchronize this value with the system. When the model reading operations are intensive, the synchronization may cause long response time on the GUI. For example, if a class has many attributes (nearly 200 ones for an extreme example GridLAB-D GUI), then when an element of this class is selected all the attributes have to be synchronized. To make it worse, due to the rendering policy of the GMF-generated property view, the visualization will retrieve each attribute three times before finally displaying the view. This can cause 10-second latency before the view is refreshed. Since many system attributes do not change so frequently, it is not necessary to update every time for every attribute. Therefore, we introduce a caching mechanism into the causal connection maintenance of runtime models to avoid unnecessary refreshment of system states.

4.2 Caching the runtime models

We introduce a caching mechanism to enhance the existing on-demand synchronization. The basic idea is to give a cached value and an expiration time for each attribute in the runtime model. After the attribute value is synchronized with the system state, subsequent reading operations to the attribute before the expiration time will be ignored. Since different attributes in the system have different changing frequencies, we support a diverse and dynamic window for expiration. We give different windows for the attributes, and keep tuning them by randomly inspecting the cache.

Algorithm 1 specifies how this caching mechanism works. When an external user, such as the visualization engine, reads an attribute on the model, instead of directly synchronizing it with the system, we first check if the previously synchronized value is expired. If it is not expired, we give a 95% possibility to directly return the cached value synchronized before, and the other 5% possibility to still synchronize in order to check if the window length is proper. Specifically,

Algorithm 1: Caching algorithm for attributes

```
Input  : The attribute attr accessed by the model user
Output: The property value returned
Global : The auxiliary mapping for all properties:
         cache, expiry, and window
curr ← currentTime()
if t < expiry(attr) then
    if under 95% possibility then
        return cache (attr)
    else
        value ← SychronizedFromTheSystem (attr)
        if value = cache(attr) then
            expiry(attr) ← curr + window
        else
            cache (attr) ← value
            window (attr) ← max(window (attr) * 0.5,
            0.1s)
        return (value)
else
    value ← SychronizedFromTheSystem (attr)
    if value = cache(attr) then
        window(attr) ← min(window(attr)*1.25, 10s)
    expiry(attr) ← curr + window
    cache(attr) ← value
    return value
```

if the retrieved system value does not equal to the cached value, then that means the system value may change more quickly than indicated by the current window time, and we decrease the window. Alternatively, if the previous value is expired, we first get the new value from the system, refresh the cache, and reset the expiry time. If the system value is still equal to the cached one, then that means the current window time is apparently not long enough, and we increase it. Note that the increasing and decreasing steps are not symmetric (0.5 vs 1.25). We decrease the window more quickly to achieve more accuracy. We initialize all the windows as 1s, which is the common frequency as the the main editor updates. We give a lower limit for window values as 0.1s, to avoid the repeated updates caused by the automated rendering algorithm of GMF. We also give an upper limit as 10s, so that the transition of an attribute from long-term inactive to active will affect the synchronization frequency quickly enough. These parameters are assigned based on GUI usage style.

We implement this caching mechanism on the Sm@rt framework. The three global maps are automatically generated and inserted into the runtime model engine, and the above algorithm is embedded into the aspect code for each attribute reading methods.

5. EVALUATION

5.1 The GridLAB-D GUI

We implement a series of GUIs for GridLAB-D simulatiors. All these GUIs have the similar appearance as the one shown in Figure 4.1 Different versions of the GUI have different types of elements to display, and the different key or detailed attributes. Among them, the trunk version supports all the 80 classes defined by the GridLAB-D modules, and 1731 attributes in total for these classes.The biggest class has 146 attributes shown in the property view. These versions have been used by members in our research group for different purposes, such as monitoring and displaying the runtime effect of the distributed learning-based optimization of residential systems, interactive tuning of the simulation systems of micro-grid, and also for new members to get familiar with GridLAB-D.

We tested the trunk version GUI by a simulator with 78 objects in total. Without hierarchical editors, all the objects are displayed on the same main editor, with totally 137 key attributes. The initialization phase takes 4 seconds in average. We set the refresh interval of the main editor as 2 seconds, and the CPU occupation is around 20% in average on a 1.6Hz Intel Core i5 CPU, 4GB memory laptop computer. For the biggest element with 146 attributes, the property view requires 1.9 seconds in average to refresh. The trunk version is an extreme example. For the common ones with less key attributes and detailed attributes of the same element, the latency is negligible. A rough comparison between the editor and the log file of GridLAB-D showed that less than 6% attributes are not synchronized in the first update after their values are changed in the system, and less than 1% are not properly synchronized after more than 2 successive updates. According to the users, the current response time and accuracy are tolerable. This result shows that the cache mechanism in works well for the GUI purpose.

5.2 The rapid development process

We develop these versions of the GUIs in a rapid and interactive way. The development process is constituted by many loops. After each loop, there is a runnable version of the GUI that has explicit difference from the previous one. Some typical and representative versions are listed as follows. 1) The *initial demonstration* is the one shown in Section 3 and 4, with three element types and five properties. In the graphical editor, all the elements are represented as simple rectangles. 2) The *residential complete* version supports all the classes defined in the residential module shipped with GridLAB-D, and is used on an existing residential simulation. 3) The *decorated residential* uses intuitive icons instead of simple rectangles to represent the elements of the residential elements. 4) The *complete version* covers all the classes defined in the default modules. 5) The *hierarchical editors* covers the classes in the residential and the transformation modules, and support "zoom-in" to see the details of a house in detail. 6) The *residential monitoring board* covering routine elements such as houses, meters and appliances, and with the physical states displayed on the main editor, such as temperature, water level, etc. Currently there are four versions (3-6) being used in the group, with many minor versions between these milestone ones.

We regard the development process as a *rapid* one mainly based on the following two reasons. First, the whole development effort was small. There was only one author of this paper in charge of the actual development of all these GUI versions, and he was not familiar with neither GMF nor Sm@rt frameworks before. Another author participated in the design and provides technical support on the two frameworks. It only took six weeks so far to achieve the four versions in use. Until now, all the development activities were around the three high level specifications as described

in Figure 1, without any manual modification on the generated code. Moreover, most part of the specification files were automatically generated: We wrote a python script to generate the meta-model from the module definitions of GridLAB-D, and used the standard GMF wizard to generate the initial version of the graph specification from the domain model. Second, each loop was finished very quickly. Some very short loops only took a few minutes, and the longest one takes no more than a week. Most of the loops only involved the modifications on one artifact, such as changing the metamodel to include new types or attributes, or changing the graph model to tune the graph representation.

6. RELATED WORKS

Runtime models are widely investigated and utilized nowadays[1]. As a dynamic and abstract representation of the system structure or behavior, a runtime model helps the system's management agents understand the runtime phenomena and control the system configuration, and are widely used to support automated management agents, such as the self-adaptation or self-organization engines. For example, Vogel et al. uses model transformation on runtime models for the self-adaptation of JavaEE systems [10]. Morin et al. leverages aspect-oriented modeling to achieve runtime model-based automated reconfiguration [7]. This paper illustrates a new way to use the runtime models, as the model-based intermediate data representation of system state for GUIs. In this way, the runtime models are used by human users, rather than automated management agents.

Data visualization is widely used on statistical information [2] and software artifacts [4]. A common approach towards data visualization is to first construct an abstract model of the raw data, and then give the graphical representation of the abstract model. In this paper, we choose the ever-changing and externally editable system state as the target for visualization, and show that the runtime models can be used as a bridge between such data and the visualized view. The work is related to the existing approaches to use modeling and meta-modeling technologies for the development of GUIs [5] and the visual language editors [3]. Instead of directly generate the whole GUI or editor code from the high-level specifications, we use runtime models as an explicit intermediate representation, separating the concerns between the system data and the graphical strategies.

Model-driven engineering is widely used to implement the rapid development of GUIs. Melia et al. [6] extends the existing web design model to support the structure modeling of rich internet application, and use the model to generation GUIs based on Google Web Toolkit. These approaches focus on the representation layer of GUIs, and thus they assume the underlying system data to conform specific formats. In this paper, we target the legacy systems with arbitrary data formats, and use runtime models to organize the data into standard format that can be directly used for visualization.

7. CONCLUSION

In this paper, we report our experience on utilizing runtime models to develop the GUI for the GridLAB-D simulation system. We discuss the role of runtime models as an intermediate representation of the running system state, in software that needs to change its appearance according to system changes. We show that with the help of existing runtime model construction and model visualization frameworks, it is possible to achieve a rapid development process based on high-level specification and automated generation. We also enhance the causal connection maintenance mechanism to suit the usage of runtime models for GUIs.

One direction of our future work is to extend the GUI support from simply graphical model editors to the general purpose GUI concepts such as menus, widgets, windows, etc. We will investigate the combination of runtime models with the existing EMF-based GUI representation approach such as Eclipse E4 [3] and Wazaabi[4], as well as the connection between runtime model elements to the components in different GUI frameworks such as SWT, Android, etc.

The current work can be also regarded as the rapid visualization of the runtime data of simulation systems. We will investigate how to generalize this idea and the implementation to the large scale runtime data of Internet of Things or other sensor-based systems.

Acknowledgment

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie)

8. REFERENCES

[1] G. Blair, N. Bencomo, and R. France. Models@ run. time. *Computer*, 42(10):22–27, 2009.

[2] U. Fayyad, A. Wierse, and G. Grinstein. *Information visualization in data mining and knowledge discovery*. Morgan Kaufmann Pub, 2002.

[3] J. Grundy, J. Hosking, N. Zhu, and N. Liu. Generating domain-specific visual language editors from high-level tool specifications. In *ASE*, pages 25–36. IEEE, 2006.

[4] R. Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):87–109, 2003.

[5] S. Link, T. Schuster, P. Hoyer, and S. Abeck. Focusing graphical user interfaces in model-driven software development. In *ACHI*, pages 3–8. IEEE, 2008.

[6] S. Meliá, J. Gómez, S. Pérez, and O. Díaz. A model-driven development for gwt-based rich internet applications with ooh4ria. In *ICWE*, pages 13–23. Ieee, 2008.

[7] B. Morin, O. Barais, G. Nain, and J. Jezequel. Taming dynamically adaptive systems using models and aspects. In *ICSE*, pages 122–132. IEEE Computer Society, 2009.

[8] H. Song, G. Huang, F. Chauvel, W. Zhang, Y. Sun, W. Shao, and H. Mei. Instant and incremental qvt transformation for runtime models. In *MoDELS*, pages 273–288, 2011.

[9] H. Song, Y. Xiong, F. Chauvel, G. Huang, Z. Hu, and H. Mei. Generating synchronization engines between running systems and their model-based views. In *MoDELS Workshops 2009*, pages 140–154, 2009.

[10] T. Vogel and H. Giese. Adaptation and abstract runtime models. In *SEAMS*, pages 39–48. ACM, 2010.

[3] http://www.eclipse.org/e4/

[4] http://wazaabi.org/