Ivette: a Modern GUI for Frama-C

Loïc Correnson

CEA, LIST, Software Safety Laboratory
PC 174, 91191 Gif-sur-Yvette France
loic.correnson@cea.fr

Abstract. Using a static analyzer such as Frama-C is known to be difficult, even for experienced users. Building a comfortable user interface to alleviate those difficulties is however a complex task that requires many technical issues to be handled that are outside the scope of static analyzers techniques. In this paper, we present the design directions that we have chosen for completely refactoring the old Graphical User Interface of Frama-C within the ReactJS framework. In particular, we discuss middleware and language issues, multithreaded client *vs.* batch analyzer design, synchronization issues, multiple protocol support, plug-in integration, graphical and user-interaction techniques and how various programming language traits scale (or not) for such a development project.

Keywords: Frama-C, ReactJS, Reactive Programming, Static Analysis Server

1 Introduction

Frama-C [1, 2] is a platform that offers mature and industrial strength static analyzers for C/C++ programs. Built around a kernel responsible for parsing, type-checking and analysis-results consolidation, the platform is extensible *via* plug-ins that can offer new analyzers, new frontends or combine existing ones in various ways. Frama-C is known to be used in education for teaching formal methods [3–5], in research projects for prototyping new static analyzers [6–8] and in industrial settings with the highest-level of certification constraints [9, 10].

Like any other state-of-the-art static analyzer, Frama-C is generally easy to use at a first glance. However, for programs of increasing complexity, it becomes difficult, even for expert and experienced users, to tune the analyzers and to understand and fix issues. Actually, Frama-C static analyzers like the EVA Abstract Interpretation Analyzer and the WP Deductive Verification Engine, produce huge sets of complex data during their computations. Sometimes, the end-user will have to dive into those complex data for investigating the source of a possible bug or an over-approximation of the analyzer, then re-start the analyzer until all problems are fixed.

To make such a development cycle practicable, Frama-C was designed from its very beginning to be accompanied by a Graphical User Interface (GUI). Instead

of running the `frama-c` command line, one can run the `frama-c-gui` command with exactly the same arguments: this will perform exactly the same computations and open the GUI in order for the user to dive into the obtained results and launch other computations interactively. Like the command-line interface, `frama-c-gui` supports extension by plug-ins, such that any Frama-C plug-in may extend the GUI with dedicated components.

Pitfalls of Frama-C's mainstream GUI

Despite the many success stories we have encountered with `frama-c-gui`, we also acknowledge many design, usability and deprecation issues. From a technical point of view, `frama-c-gui` is written in OCaml with the LablGTK bindings to the GTK graphical environment. See Figure 1 for an illustration of this GUI.

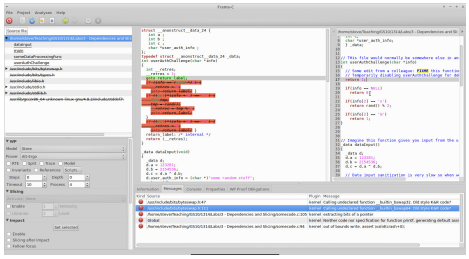

Fig. 1. The mainstream Frama-C GUI based on GTK

Although the choice of LablGTK had some advantages in the past, we have experienced many pitfalls over the years with such a design:

- Experienced developers fluent in both GTK and OCaml are extremely rare; more generally, experts in both *Static Analyzers* and *User Interfaces* are also extremely rare, despite any personal affinities in both fields.
- Developing a new graphical component in LablGTL/GTK is very difficult (*e.g.* to visualize graphs or diagrams).
- The GUI code is strongly coupled with the static analyzer. However, the GUI code is necessarily interactive (asynchronous) from a user point of view, whereas analyzer runs are definitely in batch mode (synchronous). Interleaving the two approaches inevitably produces reliability issues that are difficult to debug and fix.
- The GUI code is actually *part* of the static analyzers, which prevents Frama-C from being integrated with other user environments (*e.g.* code editors) without duplicating a lot of already existing code dedicated to the GUI.
- Last, but not least, after more than 12 years of uncontrolled hacking, the GUI code base has become completely unmaintainable.

In addition to technical problems cited above, the mainstream Frama-C GUI also has many issues from an end-user point of view:

- The user must wait before static analyzers finish their jobs *before* the GUI becomes visible.
- In case the analyzers raise errors, it is difficult to restore a safe state without exiting the GUI and restarting the entire process.
- The current implementation is cluttered by so many little bugs that using the GUI on a regular basis can be tedious.
- The existing components are not completely satisfactory, but because of the technical issues listed above, hardly nobody is akin to implement any new features in this GUI.
- The organization of the main GUI window is far too rigid, and GUI plug-ins have a very limited area available to interact with the user. This prevents designing new components and new user interactions with the static analyzers.
- With GTK, we have experienced some cross-platform differences and some difficulties with system integration: they are difficult to predict, document or even fix (*e.g.* on non-Linux platforms).

To finish on a more humorous note, we also acknowledge that using GTK has become totally old-fashioned, just have a look to Figure 1! How to hype Formal Methods, then?

Contributions

In response to the many pitfalls mentioned above, we decided to completely redesign a new user interface for Frama-C. This led to a long-term engineering effort, supported by many projects over the last 6 years. We started with a deep survey of existing frameworks, experimenting with different platforms and prototypes.

In this article, we overview in Section 2 our search for new design guidelines and the final technical stack we ended up with. We then expose implementation details of the fundamental building blocks of our new Ivette graphical user interface for Frama-C: in Section 3 we will discuss middleware and the Frama-C/Server plug-in; Section 4 will present how *React Programming* dramatically reduced the complexity of GUI code; in Section 5 we will briefly present our Dome framework of front-end components; finally we will present in Section 6 some specific features of Ivette as a *Static Analyzer* user interface. Finally, we will briefly mention future work directions in Section 7.

2 Towards a new GUI for Frama-C

Modern engines for HTML5, CSS and JavaScript are currently the most powerful means to build complex graphical user interfaces, especially for scientists: no research team will have enough manpower to re-create all the flexibility in layouts, theming, system and GPU integration, that those technologies are offering.

However, there was still an open question for the Frama-C new user interface: should it be a *web* application, a *desktop* one or an extension of some existing Code Editor ? The last option would make Frama-C bound to a particular IDE, although it is still an interesting option. However, we want to open the route to like to radically different user interactions and more *graphical* data presentations. Hence, since code editors remains intrinsically text-code oriented, we leave this option for future work and keep on an HTML-based solution.

The common idea that both *web* and *desktop* applications are the same is *wrong*! Although they can share a lot of common code, *web* and *desktop* apps do not share the same design principles and are very different from a user perspective: for instance, in *web* applications, the data the user works with must be centralized in a distant server, whereas in a *desktop* app, the user works with its own file system.

As many Frama-C developers and users, we really want a *desktop* application: we are running *possibly under development* Frama-C plug-ins on *mostly under development* source files. The co-development cycle of analyzers and analyses would be too slow if one had to connect-push-compile-and-run any modification on-the-fly. We finally ended-up with the following strong design directions:

- The GUI shall be a *desktop* application based on an HTML/CSS engine.
- The GUI shall be coded in a language that *User Interface* experts know about, with very good community support and open-source tools. This allows for taking advantage of state-of-the art human-computer interaction techniques.
- The GUI and the Static Analyzer shall live in independent processes, with middlewares and protocols to connect them with each other; this will open the route towards integration with different user environments and solve many reliability issues.
- GUI components shall be as much as possible agnostic to the precise *meaning* of static analyzer data; no complex semantic treatment over data shall be performed on the GUI side, just like Excel doesn't know what kind of data a sheet deals with, but just organizes computations over strings and numbers into cells.
- Static Analyzer plug-ins shall be ready to interact with some external User Environment, but without any dependency on the middleware that will be actually responsible for the connection. This interaction shall be coded into the Frama-C code base with some dedicated support, hence in OCaml.

Ivette Overview

Of course, there is a choice of technical platforms that are suitable for implementing such a plan. Note also that in this domain, libraries and tools are in constant evolution. The technical stack we have chosen so far consists of the following frameworks, also illustrated in Figure 2:

- A GUI desktop application using the HTML5 and NodeJS engine of the Electron [11] framework.

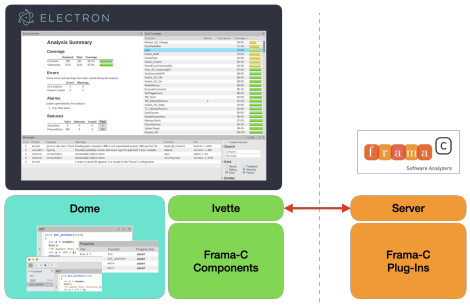

Fig. 2. Technical Organization of Ivette

- A GUI code base written in TypeScript with the *Reactive Programming* framework ReactJS [12].
- A Frama-C/Server plug-in written in OCaml, that provides an asynchronous, JSON-based, strongly typed *Request System*.
- Each Frama-C plug-in will then register new requests in the Server plug-in, independently of any communication protocol with any external User Environment.
- A collection of middlewares and protocols written in different languages to specifically connect the Frama-C/Server plug-in and the ReactJS code with each other.

The GUI code itself is entirely written in TypeScript and is split into three parts:

- The Dome framework, which is a collection of carefully designed and themed high-level components, offering a predefined choice of features, but that are robust and well adapted to end-user needs (*e.g.* tables with built-in sizing, sorting and filtering, text-editors with dynamic tag highlighting, *etc.*).
- The Ivette framework, which is an application built with Electron and Dome featuring the main Frama-C GUI graphical environment and managing the connection with Frama-C *via* the Server plug-in.
- The Frama-C plug-in components for Ivette are build from Dome components and interact with the Frama-C static analyzer plug-ins *via* registered Server requests.

Notice that since the very beginning of the project 6 years ago, some design choices have evolved, because of the technical evolutions of the frameworks over the years; and also to fix some wrong early design choices.

3 The Frama-C Server Plug-in

The Server plug-in extends the Frama-C platform by providing to every other plug-in a central way to register *Asynchronous Typed Requests*, without any

knowledge on how the server will be actually connected to external clients. The requests are entirely written in OCaml, in the same programming environment than any other Frama-C plug-in.

The Server API offers two different kinds of services: (1) a library for Frama-C plug-ins for writing and registering typed and documented requests; (2) a library for writing protocols for connecting the server to external clients. Several issues are addressed and solved by the architecture and design of the Server plug-in, notably:

- how to reconcile the sequential implementation of most Frama-C analyzers with the asynchronous requirements of a server connected to an external GUI environment;
- how to document the collection of requests registered by all Frama-C plug-ins in a consistent and maintainable way;
- how to ensure that data exchanges between the Server and the Client are well-typed, despite the fact that both programs might be developed in different programming languages;
- how to extend the server capabilities by implementing new communication protocols via external plug-ins of Frama-C.

We will present in this section the design of the Frama-C/Server plug-in and how it provides a solution to those issues. This plug-in has been open-sourced since the Frama-C v20.0 (Calcium, Dec. 2019) official release and is still in active development, although its API is quite stable.

3.1 The Request System

A discipline existing since the beginning of Frama-C is that any static analyzer shall regularly invoke a specific function, now called `Db.yield()`. In the mainstream Frama-C GUI, this function is used to periodically give an opportunity for GTK to handle user events. This is a well-known technique for implementing *Cooperative Threads* without writing the entire code of a static analyzer in any lightweight thread framework such as OCaml/Lwt.

The Server takes benefit from this old discipline to offer a similar mechanism to connected clients: each time the `Db.yield` function is invoked by a static analyzer, the server will give a chance to requests to be answered. However, the difficulty is to not interleave requests that might modify a static analyzer semantics *while* it is running.

To this end, the Server plug-in offers different kinds of requests in order to reconcile the synchronous behavior of static analyzers with asynchronous requests from external clients. Notice that all requests are initiated by the Client, whatever their kind:

- `GET` requests are meant to be very quick to compute; they can be answered at any time, even when a static analyzer is running, typically on `Db.yield` calls; they shall *not* modify the semantics of the static analyzer computations.

- `SET` requests are meant to perform analyzer configuration; they will be treated *between* static analyzer runs, hence not by `Db.yield` calls.
- `EXEC` requests are reserved for launching computations of static analyzers, which cannot be interleaved with each other and must be run in sequence. Handlers of `EXEC` requests are responsible for calling `Db.yield` periodically.

When registering a new request of some kind in the server, each plug-in is responsible for obeying the associated constraints listed above. A request actually consists of the following elements:

- the request package and name;
- a markdown documentation snippet;
- the request kind: `GET`, `SET` or `EXEC`;
- a module with signature `Input` for decoding the input parameters;
- a module with signature `Output` for encoding the output responses;
- the function, with type `Input.t -> Output.t`, responsible for answering the request.

The parameter and returned module signatures are two variants of a more general `Value` signature illustrated below:

```
module type Value =
sig
  type t
  val jtype : jtype
  val of_json : json -> t (* only for Input *)
  val to_json : t -> json (* only for Output *)
end
```

The OCaml type `jtype` encodes `JSON` types, which are a simplified version of *JSON Schemas*. Here is an illustrative extract from its definition:

```
type jtype =
  Jnull | Jnumber | Jstring | Jlist of jtype | ...
```

The Server plug-in offers a rich library for building and combining `Input` and `Output` modules. One can also declare *named* types whose documentation will be added to the request documentation.

3.2 Generic Server and Protocols

The Server plug-in provides a generic function to create a server. This function must be instantiated with a fetching function having the following (simplified) signature:

```
type 'a message = {
  requests : 'a request list ;
  callback : 'a response list -> unit ;
}
val Server.Main.create :
  fetch:(unit -> 'a message option) -> server
```

The `fetch` function is responsible for interacting with the external Client; it decodes its data into some optional `message`. A message consists of a list of input requests, together with a callback to send responses back to the client.

A very important difference between the protocol function and the plug-in request interface is that requests and responses are *not* associated in the same way. From the protocol point of view, each message consists of a list of request inputs (n, f, x) where n is some request identifier (of type `'a`), f the request name and x its JSON parameter; response messages will consist of pairs (n, y) where n is the identifier of some request from *any* of the previously received messages, and $y = f(x)$ the output JSON value resulting from processing the request (n, f, x) identified by n.

Hence, from the Client perspective, the Server is fed with requests and replies with (some of) the available responses of past requests. It is the responsibility of the Client to re-associate responses to requests pairwise and to poll the server by (possibly empty) messages until all responses are fetched back.

The current release of the Frama-C/Server plug-in comes with three server protocol implementations usable out-of-the box:

- `SocketServer`: based on `UNIX` system sockets;
- `ZmqServer`: based on the ZeroMQ [13] well-known library[1];
- `BatchServer`: this server reads static requests from JSON files and prints the responses to the terminal; this server is used for implementing unit tests for requests registered by Frama-C plug-ins. This protocol can also be used as it is for just scripting Frama-C plug-ins.

3.3 Extended Requests Features

In addition to the low-level `GET`, `SET` and `EXEC` requests depicted so far, the Server plug-in also offers more elaborated features that reveal to be very useful and intensively used in practice.

First, the Server implements *Signals* that can be used to tell the Client that something happened during a static analyzer computation. In order to save data in message exchanges, signals are only sent when the Client explicitly asks to receive them (by message type). Hence, protocol messages are extended with signal-specific requests and responses. Optionally, it is possible to associate signals to requests: hence the Client can be informed to re-issue requests when their associated signals are emitted.

Second, the Server plug-in provides so-called *Synchronized Values* which are used to automatically mirror any Frama-C internal state to the Client. Synchronized values simply consist of a combination of three basic ingredients: a `GET` request for reading the current value of the state, a `SET` request for updating the state, and a dedicated `SIGNAL` for signaling state updates to the Client.

Finally, the Server plug-in provides so-called *Synchronized Arrays* which are used to automatically mirror large *collections* of values between the Server and

[1] ZeroMQ has been notoriously used for the CERN Large Hadron Collider

the Client. This is a generalization of synchronized values, with optimizations in order to scale for huge collections: the Client is able to ask only for a limited range of records in the collection, and the Server will only send *updates* of the collection from what has already been sent in the past.

3.4 Client Side Facilities

The Server plug-in provides also facilities for building Clients.

First, the Server plug-in is capable of generating an HTML documentation of all the registered requests, with their kind, input and output JSON types and general documentation, from all the data provided programmatically by plug-ins when registering their requests into the server.

Second, the Server plug-in offers an API to programmatically browse the available requests with their type. This can be used to automatically generate well-typed JSON decoders and encoders for all requests. Typically, for the purpose of the `Ivette` GUI for Frama-C, we have developed a Frama-C plug-in that generates a strongly typed TypeScript module for each request registered in the Server plug-in. This way, Ivette GUI code can be type-checked with respect to the actual type of each request parameter and returned value, despite their low-level encoding into raw JSON data.

4 Reactive Programming with ReactJS

Although HTML5 engines are extremely powerful for their graphical rendering capabilities, developing in JavaScript and hacking the DOM object model is definitely unmaintainable for a large project.

4.1 The Language Perspective

From the language perspective itself, we decided to migrate from JavaScript to TypeScript in the middle of the project, just because non-typed languages can *not* resist refactoring: even minor API changes can simply *not* be tracked over any code base, even small ones! However, we must also mention that TypeScript comes with its own pitfalls. Namely, we have often and painfully experienced that open record typing is indeed a terrible mistake: despite being an appealing feature to deal with optional record fields, it hides all bugs caused by misspelled or renamed fields! Combined with polymorphic type inference, this makes such bugs *very* difficult to investigate and fix.

Despite those difficulties, the TypeScript ecosystem is very mature, with a lot of existing type-annotation bindings available for many popular JavaScript libraries. This is a very important requirement since we need to build upon powerful existing libraries with relatively large APIs.

The lack of available bindings for state-of-the-art JavaScript libraries is one reason for not choosing another language with a better type system than TypeScript, *e.g.* the promising ReScript [14] project. However, we still envision to

change the base language of our GUI in the future when bindings will not be a pain: even if it is a large amount of work, type safety will always make the difference.

4.2 Beyond the Model-View-Controller Paradigm

At the beginning of the Ivette project, the ReactJS [12] framework was becoming very popular for building HTML5 applications. This framework actually induced disruptive directions for *thinking* the architecture of applications, which makes GUI development *just scale* far beyond the traditional *Model-View-Controller* architectural pattern.

The following code snippet is an idiomatic example of ReactJS usage:

```
function Dlist(props) {
  const [N,setN] = React.useState(props.size);
  return (
    <div>
      <button label="+" onClick={() => setN(N+1)}/>;
      <button label="-" onClick={() => setN(N-1)}/>;
      {(new Array(N)).map((_,k) => <div>Item #{k}</div>)}
    </div>
  );
}
```

In this very simplified example, we define a new component named Dlist. The component defines a local state and, from its current value, it builds a non-trivial subtree of components, namely two buttons for updating the state and a number of items depending on the current state. Once defined, this new component can be mixed with any other standard HTML5 markup, *e.g.* `<Dlist size={4} />` ; the entire application window is built in this way.

This toy example illustrates the fundamental concept of React: the entire application window is a *purely functional* projection of the application's *internal states*, organized into components that are *function closures* that recursively build *entire subtrees* of the graphical DOM model. We now briefly comment the important terms of this statement.

Purely Functional: any React component, like the toy component `<Dlist/>` above, are *functions* that take HTML5 markup properties and render a subtree of HTML5 or Component markups. Hence, components become *first-order* citizens of the host language, like any other function closure. As such, they can be computed, stored, duplicated, partially applied and passed to other functions or components as arguments. This allows for, typically, dynamically generating entire parts of an application from both data *and* events.

Internal States: ReactJS comes with so-called Hooks [15], like the `useState()` function in the previous example. This standard hook provides the simplest possible internal state: a variable `N` initialized with the property `size` of

the `Dlist` component; this state comes with an update function `setN()` for updating the state in response to any HTML5 or programmatic events via callbacks. The developer can also create its own hooks by combining existing ones.

Components Subtree Updates: each time an application internal state is modified *via* hooks, React knows that the associated component has to be updated; this is eventually done by running again the rendering functions of each of the impacted components, and updating the concrete DOM accordingly. The magic and power of ReactJS actually lies in the amazing diffing algorithms involved in this dynamically updating process.

Compared to the classical *Model-View-Controller* paradigm, we have experienced that the main disruptive innovations that make the difference are the following ones:

- The *Model* concept is simplified by user-defined internal states powered by hooks, which are actually *part of* and *local to* each component, *together* with their associated callbacks.
- The *View* part of the paradigm has been extended to recomputing *entire subtrees* of the application graphical components, not only individual components.
- Finally, the *Controller* part of the paradigm comes almost for free: the management of component creation, deletion and layout are left to the underlying frameworks. Thanks to HTML5 and CSS layout capabilities this saves a huge amount of code compared to classical frameworks such as GTK.

Combined all-together, those major innovations allow GUI development to *just scale* for large projects. We have experienced roughly an order of magnitude reduction of the code complexity: for n components connected with each other in the GUI, the code grows in $O(n)$ complexity within ReactJS-like frameworks, whereas it reaches $O(n^2)$ complexity within GTK-like frameworks.

Last but not least, we shall also mention two *really amazing* features powered by the JavaScript environment: the first one comes from the Chromium engine embedded inside Electron, which offers a rich collection of debugging facilities for HTML5, and also direct support for ReactJS components and hooks; the second one comes from the JavaScript hot-loading features with ReactJS support, which actually allows developers to *live edit* their code: you see what you code in real time, with live interactions on your data! Compared to a traditional *compile-link-and-restart-your-analysis-to-fix-your-app* cycle of development, live-code editing provides incredibly faster development cycles.

4.3 A Quick Overview of Ivette's Hooks

For the purpose of the Ivette graphical user interface, we have typically implemented a collection of hooks dedicated to data exchanges with the Frama-C/Server presented in Section 3. For instance, given a request `rq`, one simply uses the following Hook for interacting with Frama-C:

```
const result = FramaC.Server.useRequest(rq, params);
```

The hook then automatically re-emits the request `rq` to the server each time the parameters `params` are modified or associated signals are emitted, including server shutdowns and restarts; server responses are then automatically collected and push to the `useRequest()` internal state, which will eventually make the associated components to be automatically updated.

Frama-C states and arrays (Cf. Section 3.3) also benefit from dedicated React hooks that make them automatically mirrored on demand inside the code of GUI components. The Ivette middleware actually uses internal sophisticated cache mechanisms to make all this machinery very efficient: communication between the GUI part and the Server part is super simple and smooth.

5 The Dome GUI Framework

The Electron [11] framework handles a lot of features regarding system integration and cross-platform deployment of a desktop application. However, its Chromium engine is a generic HTML5 engine agnostic *w.r.t* the *web* or *desktop* application it is actually running. It is entirely the responsibility of the developers to confer an appealing and consistent look & feel to their applications.

On another hand, there exists a huge amount of libraries available in the JavaScript community that provide interesting CSS style sheets and collections of basic components. However, they are often incomplete and complementary with each other. What is absolutely missing is a collection of all-in-one *desktop app* components designed for large real projects. This contrasts with "old-fashioned" frameworks like GTK, where you only have components consistent with each other, but it is hardly possible to create new graphical components with complex behavior, compared to the flexibility of HTML5.

Hence, we started to develop a library named Dome that consists of carefully designed and themed React components for building great *desktop* applications.

This Dome library has been designed to be re-used for User Interface projects outside Ivette/Frama-C. It is meant to be developed and maintained by GUI experts that might have zero knowledge about static analyzers and formal methods. Currently, it is still under active development driven by the Ivette needs, although we envision to open-source Dome as an independent project on a midterm basis. Here follows an overview of the already useful features currently available in Dome:

Command Line Integration: The desktop application is ready to also support command-line invocations.

User Settings: The application has built-in support for user preferences, with per-user scope and/or project scopes. Most Dome widgets have built-in support for storing their state in user settings.

Dark & Bright Themes: The desktop application supports theming *w.r.t* to system user settings.

Themed Widgets: Basic components (buttons, labels, checkboxes, icons, *etc.*) come with a consistent look & feel and consistent API.

Layout Widgets: Flexible layout containers, such as boxes, draggable splitters, grids, toolbars, sidebars, foldable panels, *etc.* with user settings support when relevant.

Icon Library: A consistent, extensible, collection of open-sourced SVG icons that do not depend on the underlying platform (necessary for documenting cross-platform apps).

Forms Widgets: A library for building hierarchical forms with support for dynamic form validation.

Hooks Library: A collection of useful React hooks to deal with events, timers, promises, etc.

JSON Library: A set of parsing operators for safely decoding JSON data in a typed way. This is typically used for User Settings, but revealed to be also useful type-safe protocol implementation such as the one of Frama-C/Server.

Drag & Drop Facilities: A library based on `react-draggable` [16] components, which offers ready-to-use Drag Source and Drop Target containers and the necessary controllers that ease the development of Drag & Drop features, which are well known to be difficult to implement. The underlying `react-draggable` library is also widely used throughout Dome.

Text & Code Editors: A library for working with large text data with arbitrarily nested semantic tags and dynamic tag highlighting. Largely based on the awesome CodeMirror [17] (v5) library, we still had to package it specifically to scale for very large texts within the ReactJS framework. Managing nested semantic tags and efficient dynamic tag highlighting typically required to hack directly with the DOM in cooperation with low-level features of the CodeMirror API.

Dynamic Tables: A library to efficiently deal with huge, filtered, sorted, dynamically loaded and updated tables. The library offers rich classes for managing the table model and customizable table views with smooth user interaction. The table views is built upon the `react-virtualized` [18] library, which offers efficient rendering of virtually infinite data sets with shadowing techniques.

6 Ivette: a Static Analyzer oriented Interface

As a graphical user interface for the Frama-C static analyzers, Ivette has some specific design orientations that, in our opinion, can be transposed to other static analyzers. The main Ivette window is organized around into a central mosaic of components the user can rearrange, picking available components from a library panel. Predefined views are also made available by Frama-C developers, and the user can also create and register their own ones.

However, all these components are *not* independent of each other. Consider for instance the `AST` component, which is responsible for rendering the abstract syntax tree of a C function after typing and normalization by the Frama-C kernel.

Each node of the syntax tree (variable, expression, statement, *etc.*) is associated to unique tags. The user intuitively expects that all other `Ivette` components will synchronize with the currently selected tag from the `AST`. Conversely, the user selecting an alarm emitted from the `Property` panel wants the `AST` to scroll to the origin of the alarm.

Hence, Ivette not only consists of independent components, but also consists of shared selection states. Currently, we have: (1) a shared (multiple) semantic tag selection state, with saved and navigable history; (2) a volatile shared semantic selection associated with mouse hover moves: each time the mouse is hovering something associated with a semantic tag, all other components can dynamically adjust their rendering.

Such a simple feature can be seen as anecdotal at first sight. However, it ends up greatly contributing towards a smooth user experience. Actually, there is always a tension when providing complex data to the user: if you provide all the available data regarding the current selection, it might clutter the limited area available to each component inside the user window. Instead, you can display a short summarized portion of the data and let the user hover this summary with the mouse, then display more detailed information for each hovered part, typically inside another dedicated component. An example of such a behavior is the `Inspector` component, which always displays a brief collection of information (that every Frama-C plug-in can extend directly *via* the Server services) related to all currently selected tags, but also for the *currently hovered* tag. This turns out to be much more comfortable for the user than, say, pop-up windows. And ReactJS is fast enough to make such a dynamic behavior very smooth.

We currently have a few basic components available for Ivette, mainly dedicated to the Frama-C/EVA abstract interpretation analyzer, that reproduce and enhance the features that were available for this plug-in from the mainstream `frama-c-gui`. We intend to add more components in a near future, namely for the Frama-C/WP deductive verification analyzer.

Some very new experimental components are also available that we could not have implemented easily inside the mainstream GTK interface: (1) an interactive exploration of the graph of EVA imprecision sources, namely the Frama-C/Dive plug-in; and (2) a dynamic pivot table data extraction of kernel and EVA results. Ivette has been open-sourced with the Frama-C v25 (Vanadium, June 2022) release.

As a general feedback from developers that contributed to the work described above, it seems that our Dome and the Ivette environments allows, even for non-GUI experts, to quickly experiment with new interaction techniques and complex data raveling.

7 Feedback and Future Work

We started to deploy experimental versions of Ivette with a few industrial and institutional partners. Initial feedback is very encouraging despite the few Frama-C components currently available. From a developer point of view, the Ivette

platform offers a very exciting and efficient environment for developing static analyzer components.

Future work will mainly focus on the development of new Frama-C components inside Ivette. We are also preparing the Dome framework for open sourcing. We want to extend the Dome framework with 2D and 3D graph capabilities, and to design components for authoring *User Documentation* directly from the GUI. There is also a need for Dome applications to support external plug-ins, in order to have dynamically installed Ivette extensions.

Another interesting direction to explore is to design Frama-C/Server protocols for other external User Environments, for instance a *Language Server Protocol* implementation.

8 Conclusion

To overcome the technical and design limitations of the mainstream graphical user interface of Frama-C, we have designed a radically different platform named Ivette. Thanks to modern technologies, namely HTML5 and CSS engines, and with the support of the disruptive *Reactive Programming* framework provided by ReactJS, we have successively reached most of our objectives.

Moreover, we managed to dispatch the necessary expertise in *User Interfaces* and *Static Analyzers* among different people: developers with strong GUI skills are dedicated to the development of Dome rich components; Frama-C developers with basically no GUI skills can still perform the hard work of implementing all necessary semantic data processing *via* Frama-C/Server requests without leaving the standard Frama-C environment; finally, those who are interested in developing new Static Analyzer GUI components can play with the Dome and Ivette environments without being GUI experts, while still producing in the end professional user interfaces for their favorite static analyzers.

The architectural design we have introduced is totally general and not dependent on Frama-C internals. The design of the Server component can be transposed to any other static analysis tool and the Dome framework can be re-used for the development of any other scientific Desktop application.

Interestingly enough, some considerations on programming language traits have been exposed and compared with each other. It is also noticeable that such a large project can not be conducted without the support of communities and open-source tools that are far beyond the capabilities of isolated research teams. We sincerely hope that this experience and feedback report will help people in the community of Formal Methods to design and build a new (hyping!) generation of Static Analyzer Graphical User Interfaces.

Acknowledgements: my very special thanks to Michele Alberti, Allan Blanchard, François Bobot, David Bühler, Maxime Jacquemin, André Maroneze, Valentin Perrelle and Virgile Prevosto for their support, valuable insights and direct contributions to this project.

References

1. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The dogged pursuit of bug-free c programs: The frama-c software analysis platform. Commun. ACM **64**(8) (jul 2021) 56–68
2. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. In: Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12, Springer-Verlag (2012) 233–247
3. Creuse, L., Dross, C., Garion, C., Hugues, J., Huguet, J.: Teaching deductive verification through frama-c and spark for non computer scientists. In Dongol, B., Petre, L., Smith, G., eds.: Formal Methods Teaching, Cham, Springer International Publishing (2019) 23–36
4. Souaf, S., Loulergue, F.: Experience report: Teaching code analysis and verification using frama-c. Electronic Proceedings in Theoretical Computer Science **349** (nov 2021) 69–75
5. Dubois, C., Prévosto, V., Burel, G.: Teaching Formal Methods to Future Engineers. In Dongol, B., Petre, L., Smith, G., eds.: Third International Workshop and Tutorial, FMTea. Volume 11758 of Formal Methods Teaching., Porto, Portugal, Springer (September 2019) 69–80
6. Many contributors: Frama-C publications on external plug-ins. `https://frama-c.com/html/publications.html#external`
7. Shankar, S., Pajela, G.: A tool integrating model checking into a c verification toolset. In Bošnački, D., Wijs, A., eds.: Model Checking Software, Cham, Springer International Publishing (2016) 214–224
8. Karpman, P.: Building up on SIDAN: improved and new invariants for a software hardening Frama-C plugin. Master's thesis, Supélec, équipe Cidre (June 2012)
9. Brahmi, A., Carolus, M.J., Delmas, D., Essoussi, M.H., Lacabanne, P., Lamiel, V.M., Randimbivololona, F., Souyris, J., SAS, A.O.: Industrial use of a safe and efficient formal method based software engineering process in avionics. Embedded Real Time Software and Systems (ERTS 2020) (2020)
10. Djoudi, A., Hána, M., Kosmatov, N.: Formal verification of a javacard virtual machine with frama-c. In Huisman, M., Păsăreanu, C., Zhan, N., eds.: Formal Methods, Cham, Springer International Publishing (2021) 427–444
11. The OpenJS Foundation: Electron, building desktop applications with HTML, JavaScript and CSS. `https://www.electronjs.org`
12. Facebook: React, a JavaScript library for building user interfaces. `https://reactjs.org`
13. The ZeroMQ Authors: ZeroMQ, an open-source universal messaging library. `https://zeromq.org`
14. Project, T.R.: Fast, simple, fully typed javascript from the future. `https://rescript-lang.org`
15. Facebook: React Hooks at a glance. `https://reactjs.org/docs/hooks-intro.html`
16. Matt Zabriskie et al.: React-Draggable, a simple component for making elements draggable. `https://github.com/react-grid-layout/react-draggable`
17. Marijn Haverbeke et al.: CodeMirror, an extensible code editor. `https://codemirror.net/5/`
18. Brian Vaughn et al.: React-Virtualized, react components for efficiently rendering large lists and tabular data. `https://github.com/bvaughn/react-virtualized`