A later version of this paper appears in *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vilamoura, Algarve, Portugal, October 2012

RobotUI – A Software Architecture for Modular Robotics User Interface Frameworks

Florian Poppa and Uwe Zimmer
College of Engineering and Computer Science
The Australian National University, Canberra, ACT 0200, Australia
{florian.poppa, uwe.zimmer}@ieee.org

***Abstract*—Modern robotics frameworks are based on modular architectures that enable them to cope with the complexity and diversity of today's robotics applications. The encapsulation of the framework modules is the key to their reuse in various robotics scenarios. Model-driven approaches further simplify the reuse of already implemented and tested modules by enabling developers to model their applications on a higher abstraction level incorporating existing modules [1]. Unfortunately, these features are only present for the implementation of the robot behavior itself, but not for the accompanying user interfaces (UIs).**

The contributions of this paper are threefold: In a first step we introduce the idea of dedicated robotics UI frameworks which allow the reuse of UIs across robotics frameworks. The paper then presents an architecture for flexible and versatile UI frameworks for robotics applications by investigating and specifying the necessary features for such systems on a platform and programming language independent basis. The introduced ROBOTUI architecture fosters code reuse on the level of self-contained UI modules and enables the user to build new robotics related UIs based on existing UI building blocks. Finally, we present an implementation of the proposed architecture that demonstrates the advantages of a dedicated UI framework and the high level of code reuse achieved by implementing the modular ROBOTUI architecture.

I. Introduction and Motivation

To cope with multi-robot systems with each robot being equipped with numerous sensors and actuators, most robotics frameworks[1] are based on a module oriented architecture and allow the distributed execution of their modules[2]. There is a large variety of such systems available today (e.g. SMARTSOFT [2], PLAYER [3], CARMEN [4], ROS [5], MICROSOFT ROBOTICS DEVELOPER STUDIO [6]). It is therefore not surprising, that one can save a considerable amount of time by using one of these frameworks and implementing the robot behavior based on already existing and tested modules for sensors, actuators and popular algorithms.

Unfortunately there is no framework available which allows the reuse of robotics related UIs in the same manner than robotics frameworks allow the reuse of their modules to implement a certain robot behavior. One might therefore ask himself: Why is it not possible to reuse the implementation of an already existing map viewer, waypoint manager or measurement viewer as a whole and not by copying parts of its source code? In addition to that, why is it not possible to simply use the same UI implementation with various robotics frameworks? And why is it not possible to create new robotics related UIs based on already available and encapsulated UI fragments?

[1]We define a *robotics framework* to be a robotics software system that implements robotics specific tools, functionality, drivers and algorithms. The system may also implement its own middleware or it is built upon an existing middleware solution.

[2]Due to different naming conventions in various robotics frameworks, the notion of a *module* is used in this paper to describe the fundamental, encapsulated unit which forms the basis for code reuse inside a robotics framework in a platform and framework independent manner.

With the increasing complexity of the robot behavior, UIs grow more and more complex as well, and their implementation and maintenance times rise. The creation of realistic 3D environments can be seen as the tip of the iceberg where the implementation often takes a considerable amount of time and also requires a fair bit of programming expertise. Up to now, such visualization modules are often implemented from scratch over and over again. This time consuming and error prone practice would not be necessary if one could use an implementation of such a module as an off-the-shelf component without any source code changes.

The key to allow this to happen in a robotics framework independent manner is to separate all modules responsible for the graphical representation and user interaction away from the robotics frameworks itself and to move them into a dedicated UI framework. That clearly draws a line between modules responsible for the robot behavior (model) and modules for the visualization of it (view). Nevertheless, the design of the UI framework also has to maintain the *separation of concerns* inside the UI framework to allow the usage of UIs across robotics frameworks. To further foster code reuse inside the UI framework, the UI implementations have to be split into reusable modules. The architecture of the UI framework has to be able to manage these UI modules to allow their reuse in various robotics scenarios.

II. Learning from Current Approaches

Having a closer look at currently existing robotics related UIs, one can spot two main ways of dealing with them: One solution is the implementation of a central UI which is tailored to the problem to be solved. The other, more common solution, is to reuse UIs provided by the underlying robotics framework.

Unfortunately, both solutions have major drawbacks. In case of a specialized central UI, the software interacts with the underlying modules of the robotics framework to gather information or to delegate user inputs. Due to the tight coupling of the robot behavior related modules and the UI modules, this very often results in a mixture of model and view. A change of the robot platform (e.g. a sensor change) or the problem itself (due to new requirements) leads most likely to extensive source code changes. This also holds true if the underlying robotics framework changes. The reuse of code between solutions of this kind is mostly restricted to source code snippets but is not possible on the level of complete, self-contained UI modules.

While the UIs provided by robotics frameworks are often very powerful, they come with a problem which is based on the module oriented architecture of the frameworks: To make the UIs reusable in many scenarios, each UI is ideally coupled to only a single robotics framework module. This results in desktops being cluttered with an increasing amount of windows (Fig. 1) in which the number of windows increases with the amount of framework modules being used (and with the complexity of the implemented scenario). Each restart of

A later version of this paper appears in *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vilamoura, Algarve, Portugal, October 2012

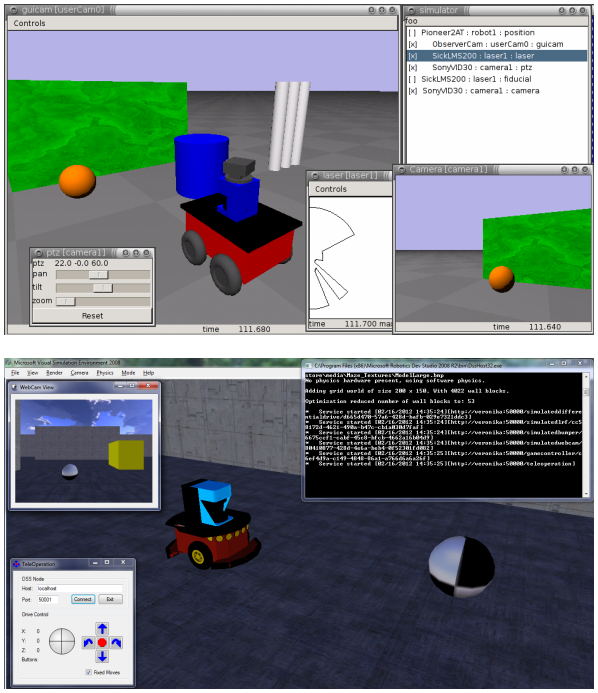

Fig. 1. Screenshot of a Player/Stage/Gazebo setup [7] (top) and a Microsoft Robotics Developer Studio setup (bottom) featuring UIs provided by the robotics framework.

either the robotics framework or the operating system results in time consuming readjustments of windows on the desktop. This waste of valuable time applies to robotics application developers as well as to end users.

Next to the problem of the missing integration of the individual UI parts into a central UI instance, robotics framework provided UIs come with another disadvantage: The reuse of their UI modules is limited to applications using the same robotics framework. That means that for example a PLAYER UI cannot be simply used by a CARMEN module and vice versa. Therefore, multiple implementations of the same UIs are available for various robotics frameworks, multiplying the overall implementation effort.

One solution to overcome this problem is to setup a communication channel between the modules of the robotics frameworks. This is hard to realize due to different and sometimes incompatible approaches of the robotics frameworks. These differences can come in form of contradicting middleware systems, programming languages, design ideas and architectures. Software bridges can solve this problem in special cases even though they are very specific and hard to maintain.

To increase the code reuse between robotics frameworks, Makarenko et al. [8] propose to implement drivers and algorithms in libraries. This enforces the clear separation between robotics framework specific APIs and the implemented logic. The libraries can then be used in various robotics frameworks, overcoming the problem of the *Software Lock-In* (also [8]).

While this library-based software reuse solves the issue for a wide range of robotics related logic and driver implementations, it does not overcome the problem for UIs. The strong coupling between UI implementations and the GUI toolkit library they are based on as well as the dependency to a specific programming language prevents the adaption of this idea.

Having a closer look onto the problem one realizes that the root of the issue is the missing *separation of concerns* inside the UI implementations. UIs could easily be used across robotics frameworks if one could exchange the source code responsible for the interaction with the robotics framework inside the UI implementations from one robotics framework to another. If each UI would have to implement such a transparent exchange and manage the interfaces to various robotics frameworks individually, the overhead would be immense. Nevertheless, if we introduce dedicated UI frameworks and move all UI implementations into it, the interfaces to the robotics frameworks can be shared among all UIs of the framework.

The introduction of dedicated UI frameworks would bring a number of advantages:

- *Easier comparisons:* Robots running different robotics frameworks can be monitored using the same UI. That simplifies direct comparisons between robotics frameworks.
- *Write once, use often:* Instead of implementing a particular UI for each framework, each UI has only to be implemented once and can be used by multiple robotics frameworks. Users profit from faster availability of new UIs.
- *Diversity:* Users can choose between UI frameworks the same way as they choose their favorite window manager in a Linux environment. They can choose the most robust implementation or a UI framework which is best suited for their specific needs. That also increases the competition between UI frameworks which can result in higher code quality.
- *Higher quality:* The amount of people interested in a specific UI implementation might be higher than it is at the moment. Problems should be exploited quicker which can result in faster bug-fixing leading to increased code quality and overall robustness.
- *Developer support:* The enforced separation of model and view in distinct frameworks frees the robotics behavior developer from any entanglement with the UI.
- *Opportunities for members of other communities:* The approach enables UI designers and developers to directly participate in robotics projects. That might result in more intuitive and usable UIs than currently available.
- *Simulation:* With a dedicated UI framework, the implemented UIs can be used both for simulated and real world scenarios. For the user it is only necessary to be familiar with one UI for both cases. In addition to that, simulators can use the UI framework for visualization purposes as well.
- *Central and distributed robot monitoring:* It is possible to either monitor a robot from a single machine or from a number of machines. If the user decides to monitor the robot in a distributed fashion, one UI framework instance would be running on each monitoring node with the same or different UIs.
- *Improved handling:* Each UI framework hosts its UIs in a single window. That makes repeating rearrangements of UIs on the desktop obsolete.

Of course, the introduction of dedicated UI frameworks will only work under specific circumstances. We discuss these requirements in detail in the following section.

III. REQUIREMENTS

A. Compatible with existing robotics frameworks

The proposed UI framework has to be able to work together with existing robotics frameworks without the necessity to change any source code in the robotics framework modules. It solely has to provide an alternative graphical representation to the existing robotics framework specific UI implementations.

A later version of this paper appears in *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vilamoura, Algarve, Portugal, October 2012

B. Side-effect free

Our definition of side-effect free means that it does not have any effects onto the behavior of the robot if the user closes the UI framework during the monitoring process of a robot. The only thing which changes is, that the user loses the ability to monitor the robot and to interact with it through the UI framework.

C. Support groups and swarms of robots

The scenarios in which robots are used today are getting more and more complex. Therefore, the trend goes from the idea of an all-purpose robot towards groups of robots where each member is specialized for a specific set of tasks. By collaborating, the group can achieve more goals than it would be possible with each robot on its own [9].

Due to the possibly heterogeneous software environment present in a group of robots, the proposed framework has to be able to cope with multiple robotics frameworks at the same time. In addition to that, the user should be able to monitor each robot on its own as well as the group or swarm as a whole. The framework has to integrate these views into a single, yet manageable central UI.

D. Modular, reusable and extendable

The content of robotics UIs is highly dependent on the task of the robot and its hardware configuration. Nevertheless, the user should not be forced to create each new UI from scratch or to copy source code snippets of other UI implementations. Instead, the UI framework has to support the user by providing robotics related UI building blocks. The user can then create new UIs based on a collection of these already implemented and tested UI fragments. It should only be necessary for the user to write source code in case one of these UI building blocks is not available yet.

In addition to that, it must be possible to reuse a UI as a whole, if the robot configurations are very similar (e.g. same sensor configuration but the parts are from different vendors) or even the same (e.g. homogeneous swarm). It is also necessary, that the proposed UI framework supports the easy integration of new UIs and robotics framework interfaces.

IV. The RobotUI Architecture

Instead of only introducing an implementation of a UI framework, we emphasize on the description of an architecture. If implemented, this architecture will result in a clearly structured UI framework allowing the user to take full advantage of the benefits outlined in section II. The platform, operating system, GUI toolkit and programming language independent nature of the architecture allows interested parties to implement their own UI framework based on their favorite combination of these. Nevertheless, we also provide an open-source implementation of the RobotUI architecture which we briefly introduce in the next section.

A. The bigger picture

As emphasized in section II, the UI framework forms an additional layer (Fig. 2) on top of robotics frameworks and their middleware systems (or any other software system deployed on the robot if no robotics framework is used). While robotics frameworks abstract the hardware, provide modules for algorithms and implement the robot behavior, the UI framework is exclusively used for user interaction and the graphical representation of the internal and external state of the robot. This is achieved by communicating with the underlying robotics framework itself or its individual modules. To maintain the *separation of concerns*, UI framework modules are not allowed to communicate directly with the hardware.

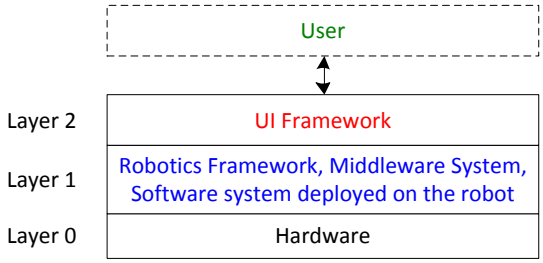

Fig. 2. UI frameworks form a new layer upon existing robotics frameworks.

There are two ways a dedicated UI framework can be implemented: One solution is that the UI framework exploits a clearly defined interface towards robotics frameworks. These can then actively push data to be visualized to the UI framework. This solution simplifies the design of the UI framework due to the fact that the UI framework does not need to know any information about the robotics frameworks connected to it. Unfortunately, this approach forces the robotics behavior developer to include source code into the robotics framework modules for the communication with the UI framework. Source code changes in the robotics framework modules would be necessary each time the interface to the UI framework changes or another UI framework should be used.

The other solution, which is implemented in the RobotUI architecture, reverses the direction of communication. The UI framework acts as an optional client to the robotics framework modules and subscribes to sensor data and user interaction requests and delegates user triggered commands to the corresponding robotics framework modules. Therefore, no UI framework specific source code is necessary in the robotics framework modules. That allows the robotics behavior developer to compose robotics framework modules without any UI entanglements. Due to the fact, that the presence of a UI framework cannot be assumed during the runtime of the robot, the robotics behavior developer is enforced to implement the robotics framework modules in a way that the safety of the robot and its environment is always ensured even though no UI framework might be available or the user does not respond in a specified amount of time. This solution therefore supports the developer indirectly by requiring him to think about these cases during the development time of the robotics framework modules.

B. The component structure

Due to the design decision to implement the UI framework as an optional client to robotics frameworks, the UI framework needs information about how it can interact with the robotics framework modules. To maintain the *separation of concerns* inside the UI framework, the RobotUI architecture has to ensure the separation of the robotics framework specific code from the UI specific code.

The main purpose of the component structure discussed here next to maintaining the *separation of concerns* is to enable the user to reuse whole UI implementations, UI modules and UI fragments with various robotics frameworks and under different robotics scenarios.

User Interface Components

A typical robot UI is composed of multiple individual parts (e.g. a window showing the current camera stream, a graph

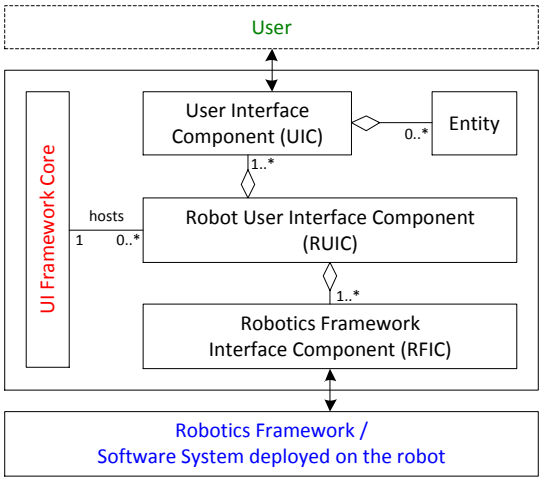

Fig. 3. The internal structure of the UI framework.

viewer). Most of these parts can be reused in various robot UIs. We call these fundamental UI parts *User Interface Components* (UIC) in the RobotUI architecture.

Each UIC provides interfaces to receive user inputs or to visualize information or both. While the granularity of GUI toolkit elements is very fine to allow them to be reused in applications of various domains, UICs are tailored to the needs of applications in the robotics domain. A UIC therefore consists most likely of a collection of GUI toolkit elements providing the user with the possibility to reuse comprehensive robotics domain specific UI modules.

A UIC can be implemented as a graphical UI, but it is not restricted to that. It is also possible to use other means of communication to interact with the user: A UIC could for example alert the user via a sound message about a critical state or implement a text-based UI.

Complex UICs can be broken down into smaller, itself reusable UI fragments which we call *Entities* (e.g. a robot visualization, an image based map, a coordinate system). Entities are robotics domain specific UI building blocks that are more comprehensive than GUI toolkit elements and form the lowest level of code reuse inside the RobotUI architecture (Fig. 4).

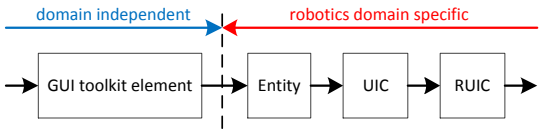

Fig. 4. Increasing granularity of code reuse in the RobotUI architecture

A UIC is not restricted to view information of only one robot at a time but can visualize the state of multiple robots simultaneously.

Robotics Framework Interface Components

In order to make UICs reusable with various robotics frameworks, we have to decouple the interface to the robotics framework from the UICs into a separate component. We call these components *Robotics Framework Interface Components* (RFICs).

Each RFIC is responsible for the interactions with a robotics framework or with one of its modules (depending on the architecture of the software system used on the robot). Therefore, the component has to implement the communication protocol required by the software framework deployed on the robot.

Robotics frameworks implement different strategies to ensure the decoupling of their modules (for example a buffer structure protected against concurrent write access). Each RFIC can also be understood as a client module of the robotics framework the UI framework connects to. The decoupling of the RFIC from the other framework modules (and therefore the decoupling of the UI framework from the underlying Layer 1) is therefore provided by the robotics framework in use.

RFICs are also responsible for the transformation of the robotics framework specific structures and messages into UI framework specific structures whenever such a transformation is necessary. Each RFIC can be seen as a bridging component between a robotics framework and the UI framework and appears transparent to the user.

Robot User Interface Components

Each *Robot User Interface Component* (RUIC) represents a UI for an individual robot or for a group of robots and consists of at least one UIC and one RFIC. The exact number of RFICs and UICs inside a RUIC varies on the task and visualization strategy of the RUIC.

A RUIC is responsible for providing a window to host its UICs, and for connecting its RFICs with the matching UICs. User inputs from UICs are then forwarded by the RUIC to the underlying robotics framework module via the corresponding RFIC. The RUIC is also responsible for managing the robot UI specific configuration.

RUICs, RFICs and UICs run asynchronously with each other. That allows different update rates for each component and maintains the loose coupling of the components.

C. UI framework core, RUIC life cycle

The core of a RobotUI framework is very compact. It solely hosts multiple RUICs simultaneously and is responsible for their handling: New RUICs can be added during runtime and existing RUICs can be removed.

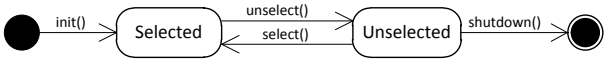

Fig. 5. RobotUI component states

The UI framework core also initiates state changes (Fig. 5) of the RUICs based on user interaction. After a new RUIC is added by the user, it will be initialized and selected. The user will only see the selected RUIC. If a different RUIC is selected, the currently selected RUIC will be unselected. That allows each RUIC to still exchange data with the robot (via its RFICs) but without the necessity to update its graphical representation. If a RUIC is deleted by the user or the program shuts down, the shutdown method of the RUIC will be called by the framework allowing the component to clean up its resources.

D. Component reusability and the importance of clearly defined interfaces and data structures

To ensure the maximum reusability of the RFICs and UICs, and to enable their transparent exchange, the following requirements must be met:

1) Internal states of both RFICs and UICs must stay encapsulated inside the components themselves and should never be passed on to other components.
2) Data structures used to exchange information between the components should be generic and use standardized units (SI units) whenever possible. The amount of data structures should be kept as small as possible. Only when the data structure provided by a RFIC is identical, or

an extended version of the data structure needed by an UIC (and vice versa), a direct connection between the components can be setup.

3) Both RFIC and UIC have to clearly define which data structures they provide and accept. Only if the interfaces of two components match, or the exchanging component has an extended version of the component to be exchanged, a transparent swap of these components is possible.

The first two rules can not be enforced by programming languages or frameworks, but are still essential to maximize the reusability of the components.

The problem of clearly defining the component interfaces is also well known in robotics frameworks that are based on modular architectures. Different ideas are implemented in these to cope with the problem: To be able to transparently exchange modules that solve the same task in different ways, PLAYER uses interfaces (generic specifications of devices) [10]. The data structures used to exchange information with the various interface implementations are specified in the interface definition. Thus, the interface creator specifies the data structures, shifting the responsibility away from the user. Unfortunately, due to the generic nature of the interface, features and functions unique to a specific interface implementation are ignored.

SMARTSOFT has a different idea to cope with the problem: Each component describes its own interface and specifies the communication objects to be used. Predefined communication objects are available and used extensively in the components provided by the framework. Due to the object oriented nature of the data structures, it is possible to derive and extend existing communication objects without loosing the generic compatibility. Unfortunately, users still have the possibility to create contradicting communication objects (same information content but different type). Such action prevents the transparent exchange of otherwise exchangeable components.

Of course commercial software tools run into the same problem. MICROSOFT ROBOTICS DEVELOPER STUDIO (MSRDS) services describe their interfaces using contracts. Other services can implement the same contract if they wish to be transparently exchangeable with other implementations of the contract. It is possible to extend existing contracts if additional functionality is provided by a service but the compatibility to the original contract should be kept. Microsoft provides a set of generic service contracts one can implement and extend. Users can also define their own, possibly contradicting, contracts for their services.

While PLAYER and SMARTSOFT use programming language and middleware specific means to describe the interfaces of their components, MSRDS introduces a programming language independent interface description. That is necessary to enable the creation of MSRDS services in all available .NET languages.

The ROBOTUI architecture does not specify an interface description language for ROBOTUI components on a programming language independent basis. Most programming languages provide means to clearly describe the component interfaces. An exchange of components of different ROBOTUI implementations is unlikely due to various incompatible programming languages and GUI toolkits. The introduction of a mandatory interface description language for all ROBOTUI implementations does therefore not gain any advantages.

V. REFERENCE IMPLEMENTATION

In this section we present the reference implementation of the ROBOTUI architecture. The open-source project is called ROBOTUI ECLIPSE RCP and is available under the URL http://robotui.sourceforge.net/robotui_ercp.

A. Technology selection

We decided to realize the reference implementation of the UI framework using Java in combination with the Eclipse Rich Client Platform (Eclipse RCP) [11], which is based upon the Standard Widget Toolkit (SWT). The decision was made based on following reasons:

- Eclipse RCP is based on Equinox, which is the reference implementation for the OSGi [12] framework specification. The standardized OSGi module runtime supports the encapsulation of RFICs, UICs and RUICs on multiple levels: Each component is implemented as an individual OSGi bundle that is represented by an individual implementation project. In addition, each OSGi bundle is packed as a .jar file on the file system. The component structure of the ROBOTUI architecture is therefore recognizable throughout the whole development and deployment process.
- The OSGi Framework supports the developer during the implementation process of the the UI framework components: It defines a modularization model for Java (Module Layer), introducing rules which enable the developer to select which contents can be seen by other bundles. A life cycle API for bundles is also provided (Live Cycle and Service Layer). In addition to that, dependencies between bundles can be described formally and version information can be attached to bundles.
- Multi-platform support due to Eclipse RCP and Java.
- Available OSGi modules can be incorporated into ROBOTUI ECLIPSE RCP.
- The Eclipse IDE offers strong support for the development of Eclipse RCP projects. The IDE is widespread and available on many platforms.
- LWJGL (Lightweight Java Game Library) as well as JME (JMonkeyEngine) can be used in Eclipse RCP plugins (= OSGi bundles), allowing the implementation of 3D environments as UICs.
- The plugin model of the Eclipse RCP allows developers to attach their copyright to individual plugins. No license model is enforced by the UI framework core, allowing developers to pick a suitable license for their plugins.

B. Example scenario

Two robots performing various tasks in a simulated environment should be monitored individually using the ROBOTUI ECLIPSE RCP framework. The robots are simulated using different robotics frameworks. The underlying maps, the robot starting positions as well as the robot configurations are different:

1) Robot *Natalie*
   - Pioneer P3DX base incl. 16 sonar sensors
   - Sick LMS200 laser range finder
   - Simulated using MSRDS
2) Robot *Keira*
   - Roomba 500 Series
   - Hokuyo URG-04LX laser range finder
   - Simulated using PLAYER/STAGE

C. Evaluation

Fig. 6 shows a screenshot of the ROBOTUI ECLIPSE RCP framework for each of the monitored robots. Even though the robot configurations and the robotics frameworks in use are different, a number of ROBOTUI components are used by both UIs (Fig. 7, black components). The robotics framework and

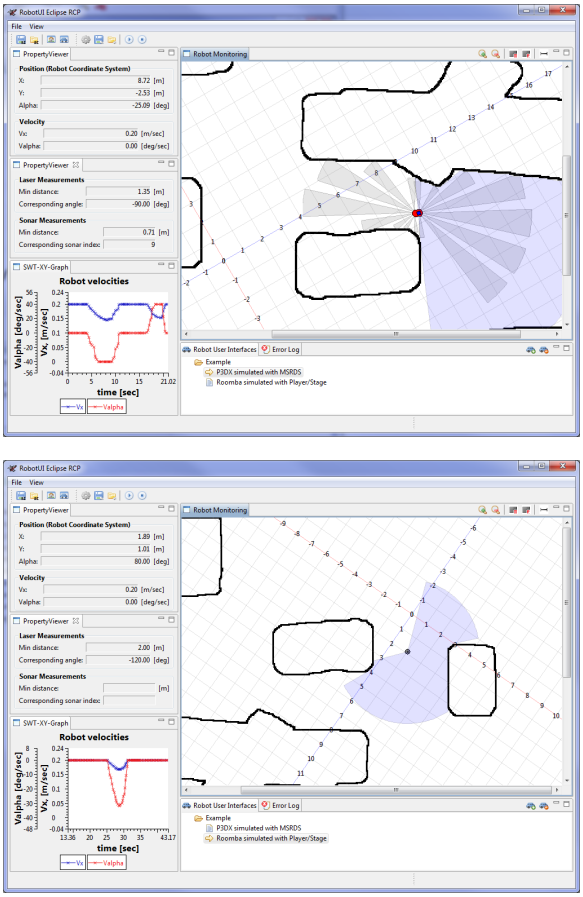

Fig. 6. Screenshot of ROBOTUI ECLIPSE RCP monitoring the robots *Natalie* (top) and *Keira* (bottom).

the robot to be used can be selected during the configuration process of the UI, and the corresponding components are created and initialized during the same process (for *Natalie* the red components are used while the blue components are used for *Keira*). Not a single line of source code has to be changed, only the configuration for both UIs are different. Due to the generic nature of the components, they can also be reused in other UIs.

VI. CONCLUSION AND FURTHER WORK

In this paper we introduced an architecture for modular robotics UI frameworks. Implementations of the ROBOTUI architecture allow the user to reuse whole UI implementations with various robotics frameworks. The UIs are composed of self-contained UI modules which can be reused in different UI implementations. Encapsulated UI fragments support the user in building new UI modules. Robotics related UIs can therefore be created and composed quickly without the necessity to rewrite source code.

We also presented a reference implementation of the ROBOTUI architecture and showed the high degree of component reuse inside the UI framework, even though different robotics frameworks and robot configurations have been used in the example scenario. The ROBOTUI ECLIPSE RCP framework is open source and freely available (http://robotui.sourceforge.net/robotui_ercp).

We are confident that a model-driven approach can be implemented on top of the ROBOTUI architecture. That would further support the user creating robotics related UIs by modeling UIs in an abstract way. The models describing UIs could also be reused between various ROBOTUI implementations.

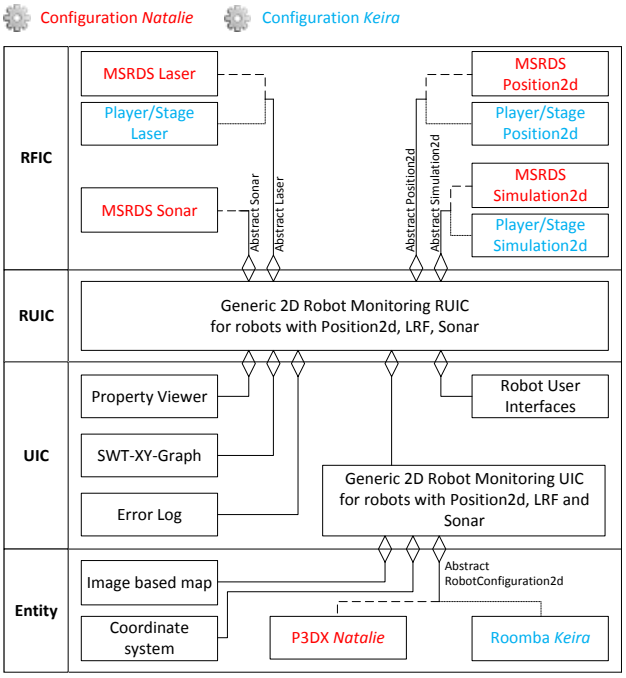

Fig. 7. Component reuse: Black components are used by both UIs. The colored components are created and initialized during the configuration process and specify the robotics framework and the robot to be used.

REFERENCES

[1] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, "Design Abstraction and Processes in Robotics: From Code-Driven to Model-Driven Engineering," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6472, pp. 324–335.

[2] (2012, Jul.) SmartSoft. [Online]. Available: http://smart-robotics.sourceforge.net/

[3] (2012, Jul.) Player/Stage. [Online]. Available: http://playerstage.sourceforge.net/

[4] (2012, Jul.) CARMEN. [Online]. Available: http://carmen.sourceforge.net

[5] (2012, Jul.) Robot Operating System (ROS). [Online]. Available: http://www.ros.org

[6] (2012, Jul.) Microsoft Robotics Developer Studio. [Online]. Available: http://www.microsoft.com/robotics/

[7] (2012, Jul.) Gazebo Screenshot. [Online]. Available: http://playerstage.sourceforge.net/gazebo/gazebo.html

[8] A. Makarenko, A. Brooks, and T. Kaupp, "On the Benefits of Making Robotic Software Frameworks Thin," in *IEEE International Conference on Intelligent Robots and Systems*, Nov. 2007.

[9] A. Ollero, I. Maza, S. Lacroix, R. Alami, T. Lemaire, G. Hattenberger, J. Gancet, V. Remu, M. Musial, G. Hommel, L. Merino, F. Caballero, J. Ferruz, J. Wiklund, P.-E. Forssn, C. Deeg, M. Bjar, F. Cuesta, L. Solaque, N. Pea, C. Nogales, F. Lpez-Pichaco, J. M. de Dios, L. M. Ribeiro, and X. Viegas, *Multiple Heterogeneous Unmanned Aerial Vehicles*, ser. Springer Tracts in Advanced Robotics, A. Ollero and I. Maza, Eds. Springer, 2007, vol. 37.

[10] R. T. Vaughan, B. P. Gerkey, and A. Howard, "On device abstractions for portable, reusable robot code," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct. 2003, pp. 2121–2127.

[11] (2012, Jul.) Eclipse Rich Client Platform. [Online]. Available: http://wiki.eclipse.org/index.php/Rich_Client_Platform

[12] (2012, Jul.) OSGi. [Online]. Available: http://www.osgi.org/