A Framework Built in .NET for Embedded and Mobile Navigation Systems

Zoltán Benedek
Budapest University of Technology and Economics
Dept. of Automation and Applied Informatics
Goldmann György tér 3, 4.em
Hungary 1111, Budapest
zoltan.benedek@aut.bme.hu

ABSTRACT

With the appearance of low cost high performance embedded and mobile computers the applications running on these computers can offer novel services on board of a wide variety of transportation vehicles. These services include providing navigation aid based on GPS (Global Positioning System) and digital maps in the field of personal in-car navigation. Considering public transportation vehicles the most important services are automatic next-stop annunciation, electronic sign control, automatic passenger counting, dispatch communication and Wireless LAN (Local Area Network) data management. Most of these applications can be decomposed into a common set of components. This paper describes a component-based framework developed in .NET to support the development of navigational applications. The core of the framework is the component configuration, component wiring and communication infrastructure which facilitates the low coupling of components and also enables the tight integration of services. Utilizing this infrastructure and building a predefined set of component building blocks the development time and cost of specific applications can be reduced significantly.

Keywords

navigation system framework, component framework, embedded event-driven applications

1. INTRODUCTION

Recently a boom in the market of general purpose personal mobile devices (Pocket PCs, Smartphones, etc.) can be observed. In addition, GPS receivers can easily be connected to most of these devices, then installing a map software (e.g. Microsoft Packet Streets) a complete navigation system can be developed. As an operating system Microsoft Mobile 2003 SE is one of the candidates targeting these devices.

Though the complete functionality is available, it is not possible to directly use these personal devices as on-board controllers on public transportation vehicles. In this field the environment calls for more ruggedized devices. Fortunately, the appearance of low cost high performance PC/104 (and other PC compatible) embeddable computers enables the application of powerful Operating Systems, such as Windows CE or Windows XP Embedded. The first public transportation vehicles equipped with on-board computers appeared quite a long time ago. Most of these systems were deeply embedded and provided relatively simple services, such as GPS based location identification and automatic next stop annunciation. Modern on-board computer systems can offer significantly wider ranges of services [Zhao97], as it will be described in details in section 2.

Applying modern managed *runtime platforms* (such as the .NET Framework or the .NET Compact Framework) on top of the Operating System these computers open new ways for *rapid application development* providing high quality integrated services. The .NET Framework is a *general purpose framework* which provides general services that are used by most applications.

This paper presents a *specialized component based framework* built on top of the .NET Framework to maximize the productivity developing embedded navigation applications. With the help of this framework the developers should be able to create

.NET Technologies'2004 *workshop proceedings*,
ISBN 80-903100-4-4

complex navigational applications offering strongly integrated high level services. These services are realized by components, the low coupling and flexible configurability of components is guaranteed by the framework according to the fundamental *design for change and reuse* concept.

The core services of this framework constitute the infrastructure that takes care of the component related (configuration, startup, communication) and the threading issues, so that application developers can fully focus on solving application specific problems. Utilizing this infrastructure the first step is to develop components, then to configure and wire them together (in an XML configuration file) according to the application specific requirements.

Although the framework currently requires the full .NET framework due to making use of a few legacy components by COM Interop, care has been taken to minimize the utilization of features that are not available in the .NET Compact Framework to ease porting in the future. The navigation framework currently targets the Windows XP Embedded platform with the full .NET Framework installed.

A remarkable amount of research has been conducted related to embedded component based frameworks [Muller01], [Doucet02]. Most of these frameworks have been developed to realize applications running on low performance embedded chips and therefore a lot of emphasis has been put on performance, memory optimization, real-time execution requirements and similar issues, which can be best achieved by unmanaged C or C++ runtime environments. As a typical example, the Balboa component framework [Doucet02] uses C++ for component development and defines a high level component integration language (CIL) that supports introspection and loose typing. Most achievements of this environment are readily provided by the .NET Framework. Instead of focusing on component integration problems in unmanaged environments the framework described in this paper targets embedded systems offering high level, complex, strongly integrated services by loosely coupled reusable components. Also, medium to high performance embedded computers are supposed to be available running managed execution environments.

2. EMBEDDED AND MOBILE NAVIGATION SYSTEMS

Public transportation systems exhibit a set of highly complex navigation applications for on-board computer systems [Bened00], [Bened04]. These systems can automate several tasks, such as making next stop announcements or driving electronic next-stop, route and destination signs. Though the vehicle position can be determined applying a GPS, in most cases *dead reckoning* capabilities based on evaluating odometer and compass information are also required to handle those cases when GPS satellite visibility is blocked by high buildings or tunnels.

Processing normal operating records on-board systems can feed a number of statistical calculations: passenger counting on a per stop basis, logging detected off-route events (detours), detecting late arrivals, early departures and alarm conditions represent a great source of valuable business information for transit authorities.

The offload of collected data and the update of the on-board route-schedule database can be fully automated if the vehicle and the garage have Wireless LAN installed and appropriately configured.

Images taken by on-board cameras can also be captured so that accidents and incidents can be played back to clarify what happened in a specific situation.

Vehicles can be connected online to the dispatch centre via some remote communication link, such as GPRS (General Packet Radio Service). Vehicles report their position and status (alarm, delay, device status, vehicle health problems) so that dispatchers can see and effectively handle fleet related problems.

As described above, a typical on-board system is connected to several back-end and front-end systems. Figure 1 depicts a whole fleet information system built around the on-board system.

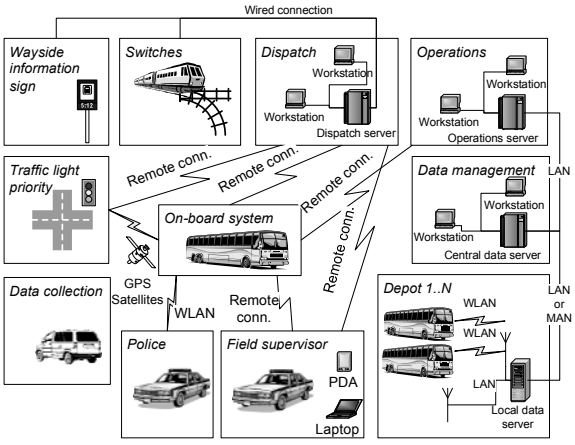

Figure 1. The system architecture of a fleet information system

There are several key points regarding the software modules running on the on-board computer. Beyond the fact that a number of tasks have to be handled by the same computer, these tasks have to work in a tightly integrated way. For instance, when the driver changes route, the new route identifier has to be has to be sent to the electronic signs installed on the

vehicle. In addition, it has to be logged into a file (source of statistical data), it has to be sent to the dispatching centre via the remote communication link and it has to be used as labeling data for the captured video frames. When developing applications the programmers have to be able to handle this complex net of interrelationships between the system modules. There is another important aspect: different transit authorities may have significantly different requirements, different vehicle infrastructures, so each software component should be easily replaceable and new modules should be easily pluggable without affecting the others.

Figure 2 illustrates the most important logical components required on board in case of a public transportation vehicle.

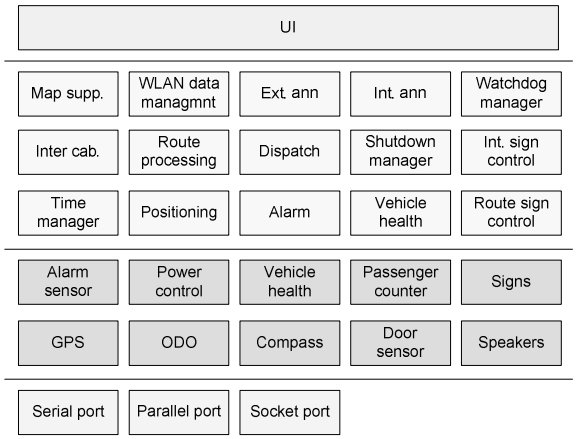

Figure 2. On-board components

In the following sections the component model of the framework, the details of component definition, component configuration, system startup, inter-component communication and threading issues will be presented.

3. COMPONENTS

Component Definition

As mentioned earlier, the framework follows a component-based approach. A component in our case has a more specific meaning than the general technical definition of most books and papers. A component conceptually is a building block encapsulating some processing logic and providing services for other components. It is a wrapper around some domain specific logic allowing the framework to treat components uniformly and to provide services for component startup, initialization, configuration, shutdown, according to the Service Configurator design pattern [Dougl99]. Components can contain an arbitrary number of objects possibly realizing very complex component logic.

Components are implemented in .NET assemblies to facilitate flexible system configuration: enabling or disabling a component will result in the given component being loaded or not loaded by the framework at startup, respectively. Components are specialized based on the layer they belong to. As a matter of fact, three layers are defined. *Port type components* (serial ports, parallel ports, sockets) form the lowest layer, they communicate with the outside world. *Device type components* abstract GPS, odometer and other devices for the higher level logic. *Application logic components* perform the real domain specific tasks. Fig. 3 shows one simple configuration made up by eight components.

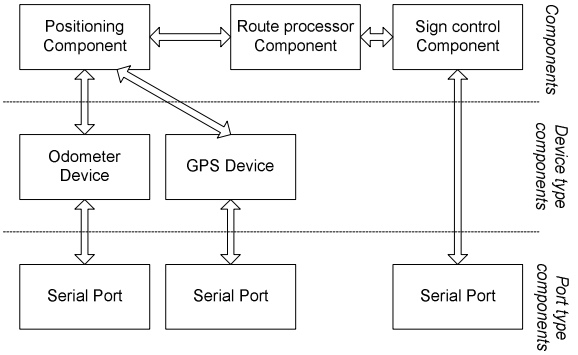

Figure 3. A simple system configuration

Component Structure

Each component has to have certain predefined interfaces and has to follow some predefined rules so that it can be managed by the framework. Components are composed of three parts as shown in Figure 4.

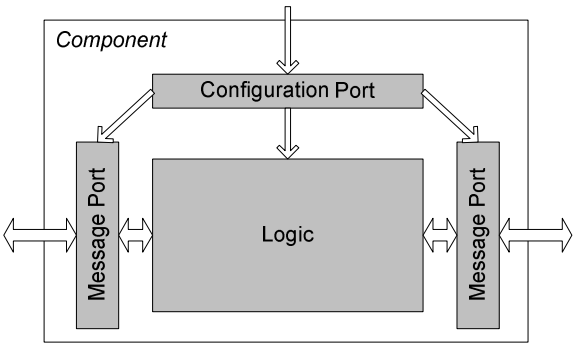

Figure 4. Component structure

The *Logic* part represents the objects performing the domain specific tasks the component is responsible for.

The *Configuration Port* is the connection point to the central Component Manager object, which is the central configuration management unit of the whole framework with respect to component startup, shutdown and configuration. The ConfigurationPort class provides built in support for the component for

starting/stopping/restarting and configuring based on a configuration file section belonging to the component. The component developer has to derive a new class from the ConfigurationPort class. This class has a CreateSubComponents method, which shall be overridden by the component developer to instantiate the objects internal to the component.

The *Message Port* objects implement message-based communication with other components, as it will be described in more detail later on. They basically receive messages from components they are connected to and forward messages to the Logic part. They also transform events raised by the Logic part to messages to be published to other components.

Although the development of a new component may look complicated, all what a component developer in most cases has to do (besides writing the component logic code) is derive two classes from the ConfigurationPort and MessagePort classes and override some of their methods.

4. COMMUNICATION

Conceptual Considerations

The cornerstone of being able to create applications that are manageable and extendible despite the complex interrelationships between their components is to define a communication infrastructure that yields low coupling between components. Instead of using strongly typed interfaces, the communication primarily based on events following the publisher-subscriber pattern results in a far more flexible solution. Taking this into consideration an event driven approach following the push model [Szyper98] has been chosen, no direct method calls are used.

The push model communication concept naturally suits the event-driven nature of the embedded application domain the framework has been primarily designed for. The representative configuration shown in Figure 3 is a good example. In most cases the GPS device calculates and sends new position information once in every second. This data is received by a serial port component, which raises an IncomingData event ("event" means conceptual events and not .NET events in this case). The GPS Device component receives this event as it is registered at the Serial Port component. It analyses the data and extracts the position information according to the communication protocol of the GPS device. The position information is sent to the Positioning component, which possibly checks GPS coordinate validity based on data received last from the odometer device and sends the noise filtered position to the Route Processor component. Next, the Route Processor checks if the vehicle has entered/left a close proximity of a stop and rises appropriate events. The Sign Control component is registered to stop change events and updates the electronic signs according to a specific sign communication protocol.

This approach has several advantages. First it inherently supports the broadcasting and multicasting of events. When the components are developed it is not known which other, not yet existing components will be interested in their events. This is not an issue if events are used. The subscription schema is defined in an XML configuration file processed by the framework at application startup. Even though this configuration file has to be edited manually now, an application with an intuitive user interface is being developed to help creating wiring definition.

Events can be implemented as sending and receiving *message objects*. In this case the parameter of a callback (event handler) method is always an object that can be perceived as a message encapsulating the type and the parameters of the event. This approach enables the logging of these messages to a log file during in the field operation, which can be played back in simulation mode later on. Multithreading combined with message queues yields a solution that can handle communication with slow hardware in a separate thread and also can hide threading issues from the programmer. Furthermore, messages can be easily serialized and sent via sockets making communication transparent across process or machine boundaries.

However, there are some liabilities. Realizing calls as sending messages makes calls requiring result more difficult to handle, as the correspondence between a particular request and a particular response has to be handled explicitly by the programmer.

Implementation

In the navigation framework inter-component events are implemented as sending and receiving message objects. In fact, messages are parameters of .NET delegates. When a .NET delegate is fired, the registered objects receive the message as the parameter of the event handler method. The type of event is encoded by the type of the message parameter. Therefore, there is no need to define a separate .NET event member for each event type, new message types can be introduced without modifying the interface of the related classes. The set of services offered and the events published by a component can be represented by different command and status type messages, respectively. There is a class hierarchy of messages with the IMessage interface as the root. Figure 5 shows a few message types involved in GPS communication.

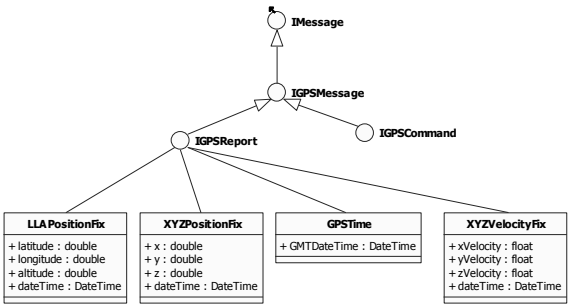

Figure 5. A part of the message class hierarchy

Each component has at least one Message Port object at the component boundary as mentioned earlier. This acts as a communication gateway between the logic part and other components. It has an OnNewMessage event handler method with an IMessage parameter, which is a reference to the message object received. The OnNewMessage method is registered (via .NET delegates) at other components and is called when there is a new incoming event for the component. The registration of this event handler at the event source components is performed by the framework at system startup. It is the responsibility of the component developer to "translate" incoming events to appropriate method calls into the component logic. The component developer also has to decide which events originated by the logic part should be transformed into messages and then forwarded to other components through the Message Port.

The core framework objects constituting the infrastructure services handle messages only through the IMessage interface, so they do not need to be recompiled when new components with new message types are introduced in the framework.

Messages are arranged into a class (or type, as interfaces are also involved) hierarchy. A few examples of different message types are the following: GPS related messages (e.g. position and time), position messages (either GPS or calculated from odometer and compass data), route processing status messages for detected off-route and late arrival conditions.

Filtering incoming events based on the type of the incoming event is possible in the Message Port. The following code snippet illustrates a component that is interested in GPS report messages only and ignores all other messages:

```
protected override void NewMessage(IMessage
message)
{
   if (message is IGPSReport)
        NewGPSReport(message);
}
```

The NewMessage method is an abstract method defined in the MessagePort base class. The OnNewMessage event handler calls the NewMessage operation for each incoming event. This way the component developer is forced to override the NewMessage method in his/her MessagePort derived class and handle the events appropriately.

The next piece of code illustrates how the MessagePort "derived" object translates the incoming message to a method call into the component logic.

```
protected override void NewMessage(IMessage
message)
{
   if (message is LLAPositionFix)
   {
        theComponentLogic.InLLAPositionPacket
        (message as LLAPositionFix);
   }
        …
}
```

If in case of a specific component the same object is the target of all incoming events it is possible to free up the programmer from writing any dispatching code. The name of the method to be called can be derived from the name of the type of the message object parameter according to a certain naming convention. First the existence of the method should be checked. If the method exists then it can be called with the message as a parameter. The next code fragment realizes this simple algorithm:

```
protected override void NewMessage(IMessage
message)
{
   string methodName = "On" +
        message.GetType().Name;

   MethodInfo minfo=
   theCompLogic.GetType().GetMethod(met
   hodName);

   if (minfo!=null)
        minfo.Invoke(theCompLogic,
        new object[] {message});
}
```

Mainly the receiving aspect of communication has been discussed so far. With respect to sending events to other components the developer of a component can call the dispatchMessage(IMessage message) method of the MessagePort class to send events to other components.

5. COMPONENT WIRING

The communication model of the framework supporting the publisher-subscriber pattern does not provide by itself a comprehensive solution. If the definition of component wiring - connecting subscribers to publishers - is awkward, then building specific target applications remains difficult.

Therefore, the configuration of components with the wiring information can be defined in an XML configuration file. Each component has a section within this file with a unique name, the assembly

name containing the code of the component, the class name of the Configuration Port object of the component, the enabled/disabled state, the configuration data specific to the component, and a ComponentConnections section with the list of the names of the components whose messages the component subscribes to. The next section shows a fraction of a sample configuration file.

```
...
<Component>
   <UniqueName>Route Processor</UniqueName>
   <Assembly>RouteProcessor</Assembly>
   <Class>RouteProcessor.RouteProcConfPort</
      Class>
   <Enabled>True</Enabled>
   <ComponentConnections>
      <ComponentConnection>Positioning
         Component</ComponentConnection>
      <ComponentConnection>Main
         UI</ComponentConnection>
      </ComponentConnections>
   <AutomaticPathStart>True
      </AutomaticPathStart>
</Component>
<Component>
   <UniqueName>Positioning
        Component</UniqueName>
   <Assembly>PositioningComponent</Assembly>
   <Class>PositioningComponent.PositioningCo
        mponent ConfPort</Class>
   <Enabled>True</Enabled>
   <ComponentConnections>
      <ComponentConnection>GPS Device</
        ComponentConnection>
   </ComponentConnections>
</Component>
...
```

The framework processes the configuration file at startup. The next code sample shows how simple .NET reflection makes the dynamic loading of assemblies and the creation of objects, given only the name of the class as a string:

```
…
ComponentListElement cle;
Assembly assembly = Assembly.LoadFrom(
  cle.assemblyName + ".dll");

Type type = assembly.GetType(
        cle.className);

cle.componentRef =
(ComponentConfManager)Activator.CreateInsta
        nce(type);

cle.isLoaded = true;
```

The ComponentListElement is a simple class holding the name of the class to be instantiated, the name of the source assembly and a reference to the created object.

6. SYSTEM STARTUP

The navigation framework instantiates a singleton Component Manager object at application startup, which takes care of most of the component loading, instantiation, configuration and cleanup tasks.

The Component Manager processes the system configuration file, instantiates the Configuration Port object of the enabled components based on their assembly and class name, and stores a reference to them in a running component list. Having the Configuration Port object instantiated its CreateSubcomponents method is called. This method has to be overridden by the component developer to instantiate the objects internal to the component. In the next step the Component Manager calls the Configure method on the Configuration Port object of the component (passing the XML path to the section of the configuration file belonging to the component) so that the component can configure itself. The same steps are performed for each component in turn. At this point the Component Manager registers the OnNewMessage event handler method of the subscriber Message Port objects at the event source Message Port objects according to the component wiring schema defined in the configuration file. Now the runtime configuration has been set up and the communication between components is enabled. The framework calls the Start method of the Configuration Port objects, which can be overridden by the component developer giving a chance to perform startup tasks specific to components.

7. THREADING

Most components perform tasks that do not take a long processing time and therefore can run in the main thread of the application without involving a thread context switch. Most *port type* components (e.g. serial ports) however have to perform long running blocking operations, for example reading data from a device. These operations can not be performed in the main thread as it would halt all other components running here. Consequently, port type components inherently have to utilize multithreading. However, most application and component developers do not have experience in handling multithreading issues (applying mutual exclusion locks properly and avoiding dead locks). For these reasons the framework has built in threading support for port type components. Each port type component starts a separate thread to perform blocking operations and hides it from the component developer.

The key issue is how to dispatch events to components connected to the port from within the *main thread* (for instance when new data is received). If the events *originated from the external thread* of the port can be routed by the port to itself in a way that they are *triggered to other components from within the main thread*, then from this point the intra-thread event dispatching mechanism (discussed in section 4) can be used. The key question is how to

"inject" the call into the main thread for the specific port object. .NET delegates form the bases of the solution as they encapsulate both the target objects and the method to be called.

Each thread (including the main thread) creates a synchronized message queue in its TLS (Thread Local Storage) memory area to hold references to messages sent by other threads. Figure 6 outlines the solution.

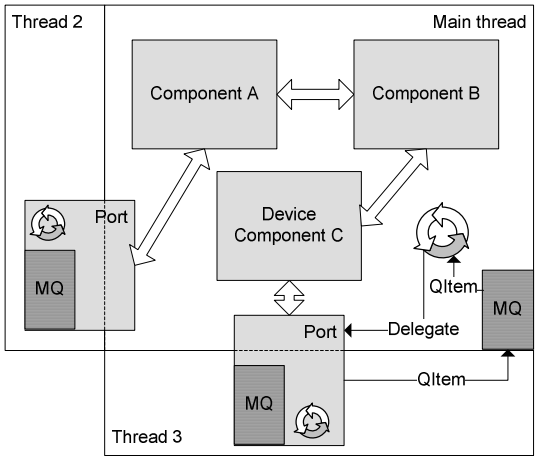

Figure 6. Threads and message queues

When new data is received by the port, an object of class QueueItem is created (abbreviated as QItem in Figure 6) by the external thread. This class encapsulates the delegate to be fired with its parameter, which is always of the type IMessage.

```
public class QueueItem
{
   public NewMessageEvent messageDelegate;
   public IMessage message;
};
```

The *messageDelegate* member is set up by the component, so that when the delegate is fired a specific method (dispatchMessageHandler) of the *sender* component is called. The *message* member is set to the message object parameter, for instance to and object of class IncomingData in case of a serial port component that has just received new data. The newly created QueueItem object is put into the message queue of the main thread. The main thread taking out the QueueItem object from its queue fires its *messageDelegate* member, passing the *message* member as a parameter:

```
queueItem.messageDelegate(queueItem.message)
```

This results in calling the dispatchMessageHandler event handler method of the *same* Message Port object from within the *main thread*. The dispatchMessageHandler dispatches the event to all registered components exactly the same way as sending events to components residing in the same thread, making threading fully transparent for the *receiving* component. Sending messages across thread boundaries is also made transparent for the *sender* component. The developer of a port type component calls the dispatchMessage(message) method in the MessagePort derived object to send messages to other components. This method is inherited from the MessagePort base class and is implemented differently for port type components, which use a message queue, and the regular components, which use direct invocation for message dispatching.

The scenario is very similar with respect to the flow of events in the opposite direction. If data is sent by a higher level component to a port type component, the event with the message parameter is transparently dispatched from the main thread to the message queue of the port, and from the queue to the appropriate handler function.

A different approach would be using .NET asynchronous delegates. Calling a delegate asynchronously results in executing the method encapsulated by the delegate by a thread pulled from a thread pool. This solution however would require the called method to take care of the synchronization by using appropriate locks when accessing shared data, and threading issues could be not hidden by the framework.

It should be mentioned that the .NET Framework has built in support for message queuing based on the COM+ MSMQ facility, made available under the Messaging namespace. The MSMQ targets enterprise applications, and is less suitable for performance sensitive embedded systems. The message queue implementation of the navigation framework uses the System.Collections.Queue class and the synchronization for the queue is provided by the System.Threading.Monitor class, together yielding a simple and efficient implementation.

8. SIMULATION

One of the most appealing benefits of message based inter-component communication is that messages can be serialized to a log file during in the field operation. This log file can be played back later on in simulation mode enabling the full reconstruction of system behavior. This feature can be really useful for instance in the fine tuning phase of developing dead reckoning algorithms, or when an accident occurs and the exact scenario has to be played back with an enhanced user interface representation (e.g. with the vehicle symbol animated on a digital map). Only messages created by *port type* component are logged as these are the components that communicate with the outside world. Each message is logged with the

name of the component that just generated the message. During simulation the simulation manager object reads the log file, extracts the messages and forwards them to that particular component which originally generated the message. According to this fundamental design concept applications can be started either in normal or in simulation mode. Most components are unaware of the actual mode, only port type components have to able to be switched to normal or simulation mode. In simulation mode the port type components are supposed to disable their hardware connections and dispatch data sent by the simulation manager.

9. SAMPLE APPLICATION

The infrastructure services of the framework have been implemented in the .NET Framework. Besides, the following components, which can be used as building blocks for developing applications have been developed so far: Serial Port, GPS Processor (handling Trimble TSIP protocol), Positioning and Route Processing. The Route Processing component encapsulates relatively complex component logic: it is fully capable to determine the relative position of the vehicle according to the route number set by the driver, can trigger next stop announcements and similar events. A map framework using an appropriate digital map database has also been developed within the ESRI MapObjects map component framework, which was built into the user interface component to visualize vehicle location, vehicle status and the current route with all the stops along that route. To demonstrate system capabilities an application has been composed using these components. Figure 7 illustrates the application:

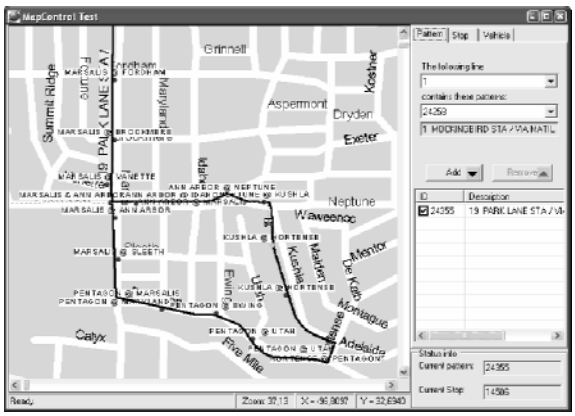

Figure 7. An application built utilizing the framework

10. CONCLUSION

A framework effectively supporting the development of complex, highly integrated, easily extendible applications in event-driven environments has been presented in this paper. As a primary application field navigation on-board computer systems have been discussed. When creating application building blocks developers can easily encapsulate system logic into components. Once the components have been developed, creating specific target applications from these components is straightforward. A communication mechanism has been elaborated following the push model: events are dispatched by the framework from the source component to each component connected to the source according to the component wiring schema. The wiring itself is defined in a configuration file, along with the individual settings specifically required by the given component. Future work will include hiding objects common to all components (Message Ports and Configuration Ports) from the component developers and enable them to provide these as .NET attributes. This will further simplify the component development process.

11. ACKNOWLEDGEMENTS

The author would like express his thanks to István Bihari, Balázs Osztatni and Milan Trenovszki for their valuable contribution in designing and implementing both the framework and the building block components.

12. REFERENCES

[Bened03] Z. Benedek: Intelligent Integrated Fixed Route Transportation Systems, Wesic Conference, 2003

[Bened04] Z. Benedek, S. Juhász: Intelligens Járműfedélzeti Rendszerek, Elektrotechnika folyóirat (Intelligent On-board Computer Systems, Journal of Electronics), 2004

[Doucet02] An Environment for Dynamic Component Composition for Efficient Co-Design, In Proc. Design Automation and Test in Europe, 2002

[Dougl99] S. Douglas et al.: Pattern-Oriented Software Architecture, Volume 2, John Wileys & Sons, Ltd, 1999

[Richt02] Jeffrey Richter: Applied Microsoft .NET Framework Programming, Microsoft Press, 2002

[Szyper98] C. Szypersky: Component Software – Beyond Object-Oriented Programming, Addison-Wesley, New York, 1998

[Muller01] Muller, P.O. et al.: Components @ work: component technology for embedded systems, Euromicro Conference, 2001

[Zhao97] Y. Zhao: Vehicle Location and Navigation Systems, Artech House, Inc, 1997

[Web01] ITS Online, <http://www.itsonline.com>

[Web02] The United States Department of Transportation, <http://www.dot.gov>