A Context-Aware Reflective Middleware Framework for Adaptive Real-time Vehicle Applications

Shengpu Liu
Lehigh University
19 Memorial Drive West
Bethlehem PA 18015
610-758-4801
shl204@lehigh.edu

Liang Cheng
Lehigh University
19 Memorial Drive West
Bethlehem PA 18015
610-758-4801
lica@lehigh.edu

ABSTRACT
Software has become crucial to develop vehicle systems. Future unmanned intelligent vehicle safety systems will increasingly rely on situational contexts collected at runtime through temporally built ad-hoc and dynamic networks for vehicle-to-vehicle and vehicle-to-roadside communications and dynamic adaptation to the contexts to improve vehicle safety and reduce traffic congestion. Context-aware reflective middleware, which can measure real-time contexts and accordingly reconfigure the behavior of supported applications, is an important technique to enhance the affordability, flexibility, and adaptability of the future vehicle safety systems. However, the long reconfiguration time of existing context-aware reflective middleware cannot satisfy the stringent real-time requirement of the vehicle systems and thus limits its adoption.
In this paper, we present MARCHES, a context-aware reflective middleware framework, which improves the reconfiguration efficiency for engineering adaptive real-time vehicle applications in dynamic environments. Different from traditional single component-chain based middleware, MARCHES supports an original structure of multiple component chains to reduce local behavior change time. Further, according to the new structure, a novel synchronization protocol using active messages is proposed to reduce distributed behavior synchronization time. Experimental results show that the reconfiguration time of MARCHES is reduced from seconds (s) to hundreds of microseconds (μs). Evaluations demonstrate that MARCHES is also robust and scalable and generates small memory footprint, which makes it suitable for supporting real-time vehicle applications.

Keywords
Reflective middleware; vehicle applications; real-time adaptation.

1. INTRODUCTION
Vehicles have been one of the most important tools in modern life to provide us transportation and convenience and even evaluate the productivity of a country. According to a 2006 DOT study, there were about 250 million registered passenger vehicles on the road in US, and approximately 16 million new vehicles are sold every year [7]. On the other side, the abundance of vehicles leads not only to congestion and energy waste but also to serious traffic accidents. On an average, there are more than 6 million car accidents on the roads of the US annually. More than 3 million people get injured due to car accidents, with more than 2 million of these injuries being permanent and more than 40 thousand deaths every year [8].

To improve the vehicle safety and efficiency, researchers have developed many electronic and digital assistant systems in this area, like the OnStar telematics system, Google Earth, and Microsoft's Windows Live Local [9]. These systems can provide helpful information to automotive drivers and assist them to make correct decisions. However, they still require the participation of human, and most of the provided information is static or delayed, and thus they are lack of the autonomy to emergent accidents, which are the major cause for people injury.

Recent advances in wireless technologies, like IEEE 802.11 and Dedicated Short Range Communication (DSRC), smart sensing, and control systems have led to the flourish of intelligent vehicle safety applications, which are designed to create autonomous, self-organizing, and mobile wireless ad-hoc networks connecting vehicles and roadside infrastructure with each other and integrate these networks with vehicle-control systems, like automatic engine, transmission, and braking systems for context gathering and avoidance of dangerous scenarios. These Vehicle Safety Communication (VSC) techniques help share situational contexts, like the actions of nearby vehicles and the situations of road-side infrastructure based on vehicle-to-vehicle and vehicle-to-roadside communication. Along with the development of VSC techniques, a lot of vehicle safety applications are also proposed to provide the real-time contextual information to drivers and warn them in critical situations or autonomously react to such situations to avoid accidents through on-board equipments.

As the VSC technologies and VSC supported vehicle safety applications are still being developed worldwide, certain particular challenges that obstruct or delay their further advances in this radically new realm also emerge. The first challenge is affordability. Although there are many VSC projects have been initialized, like the VII [1] and CICAS [2] in US, AHS [3] and DSSS [4] in Japan, and C2CCC [5] and PReVENT [6] in Europe, most of them have their individual communication standards and supported safety applications built entirely from scratch, which makes their development costs very high. The second challenge is flexibility. Existing multi-faceted and widely explored VSC projects lack flexible capable of supporting different communication protocols and new technical standards, e.g. VII uses DSRC as its communication standard while C2CCC only supports IEEE 802.11, which makes it difficult to communicate between a car installed with DSRC equipments and a car installed with C2CCC equipments. The monolithic application structure also increases the application update cost. The third challenge is adaptability. The communication between vehicles or vehicle and roadside will be more complex and with higher overload in future,

for example, PReVENT plans to support 2D/3D image capture and transmission, which can help drivers or intelligent vehicles share visions for action replan. On the other hand, the temporally constructed ad-hoc networks among vehicles and roadside infrastructure are volatile, while existing VSC techniques are lack of adaptability to the changing contexts at runtime, which affects the communication robustness and required QoS, like latency, jitter, and security etc., due to the high speed of vehicles. We argue that all these challenges have prevented vehicle safety applications from fully taking advantages of flourishing VSC techniques and then motivate to introduce an important software technique, context-aware reflective middleware, to solve these challenges and support future intelligent vehicle applications.

Context-aware reflective middleware techniques are favorable for vehicle applications for the following reasons. First, middleware, as gluing software between applications and underlying operating systems and networks, can abstract low-level implementation details and heterogeneity and then facilitate the implementation of complex vehicle applications so that developers can pay more attention to the application logic and architecture design. Second, reflective middleware uses component-based metamodel to enables reusable service components to be organized, configured, and deployed to develop vehicle applications, which consists of a component chain or a functional path, efficiently and robustly. (For example, the minimal meaningful vehicle safety application consists of a communication component and a warming or control component). Therefore, it can reduce the cost and complexity of software upgrades by incorporating new technique and protocol components and improve application flexibility. Third, context-aware reflective middleware can automatically measure and evaluate real-time situational contexts, e.g. the network conditions, hardware resources and application QoS, and dynamically reconfigure the application behaviors to adapt to the changing contexts at runtime. Therefore, it improves the adaptability and communication robustness of vehicle applications.

To better clarify the motivations and potential advantages behind the context-aware reflective middleware for vehicle systems, we present an example of possible use case (see Fig. 1). Suppose a road has two lanes in one direction, on which *car* 1 and *car* 5 are on the *lane* 1 and *car* 2, 3, 4, and 6 are on the *lane* 2. There are two scenarios that a vehicle system may need to adapt its behaviors to real-time contexts.

The first scenario is for robust communication. *Car* 1 and *car* 2 share their visions by exchanging image data for action replan when they drive closely while both only have partial vision of the road condition. Each image frame is separated into tiles and transmitted in a sequence based on different priorities. The tiles closer to the interest point have higher priority and will be transmitted first with high image quality. However, the network condition, e.g. the bandwidth, between *car* 1 and *car* 2 is dynamic and volatile. The middleware can automatically measure the bandwidth and adaptively reconfigure the compression behaviors at runtime, e.g. using or not using compression component, or setting varied compression ratio, to satisfy the required QoS, like the specified transmission time, in the application while provide images as clear as possible.

The second scenario is for action replan. *Car* 4 finds that it is too congested to drive on *lane* 2 while there are much less number of cars on *lane* 1 through the communication with nearby cars and roadside infrastructure. It then needs to switch to *lane* 1 to reduce traffic congestion. The middleware in this scenario will automatically collect the position and speed information of neighbor cars and the road conditions and then make the decision of switching to *lane* 2 by adjusting the direction and speed parameters of its software control components.

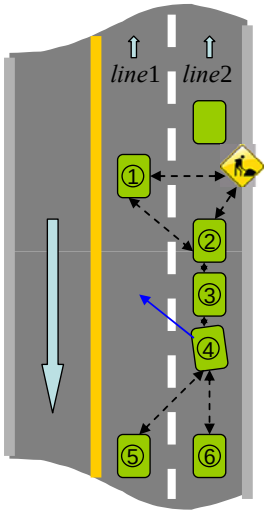


Figure 1. An example of vehicle application scenario supported by the context-aware reflective middleware.

When we applied existing context-aware reflective middleware techniques to vehicle safety applications in our experiments, we found that the existing techniques could not satisfy the stringent real-time requirements of vehicle applications, in which a minor delay may result in critical accidents and loss of lives and properties, due to their long reconfiguration time. This is resulted by the inefficiency of their synchronization protocols. Synchronization is a critical process for reconfiguring a networked vehicle system that consists of multiple programs distributed on different vehicles. Because each program has its own behaviors and local architecture, it is important to coordinate the reconfigured behaviors to achieve global behavior consistency. For example, changing or adding an image compression component in a local program may require a corresponding change or insertion of the decompression component in receiver programs in the scenario 1 of above example; reconfiguring the direction or speed of a vehicle requires corresponding action changes of other related vehicles to avoid collision in the scenario 2. Synchronization protocols are proposed to address the coordination problem so that a local program can dynamically synchronize its behaviors at runtime to the changed behaviors of other distributed programs without any prior knowledge. However, the synchronization process of existing context-aware reflective middleware is synchronous that requires the synchronization participants to be blocked until the reconfiguration process is completed. As a result, the reconfiguration time is normally in a range of seconds or even tens of seconds and affected by the network conditions and the availability of other synchronization participants. The long reconfiguration time limits the adoption of context-aware reflective middleware by vehicle systems.

In this paper, we propose MARCHES (Middleware for Adaptive Robust Collaborations across Heterogeneous Environments and Systems), a context-aware reflective middleware framework, to solve the critical issue of the reconfiguration time and to engineer time-critical vehicle systems in dynamic environments. Compared to the traditional middleware that supports the single component-chain (Fig. 2a) based application architecture, MARCHES maintains multiple component chains (Fig. 2b). Therefore, there is a new method proposed for the behavior reconfiguration that

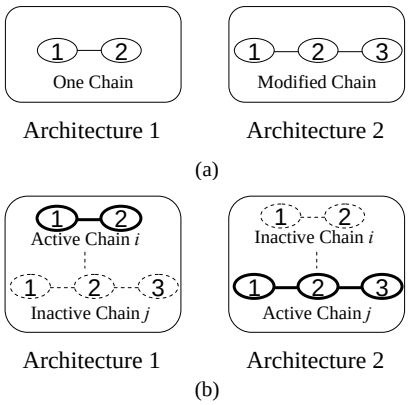


Figure 2. Dynamic reconfiguration architecture: (a) single component-chain architecture in existing middleware, (b) multiple-component-chain architecture in MARCHES.

switches active and inactive chains. This new method replaces the traditional method of modifying the single-chain structure to reduce the *local behavior change* time. Further, according to this method, an efficient active-message based synchronization protocol is proposed for asynchronously coordinating the behaviors of distributed programs. The key idea of the protocol is that each application-layer data packet takes an active-message header that indexes the correct component chain of the packet receiver to process the data payload. Therefore, the *distributed behavior synchronization time* is also dramatically reduced by eliminating the operation suspension time and buffer clearance time. The robustness of the distributed application is improved since the use of active messages results in no synchronous communication and system halting in the synchronization process. The costs introduced by this improvement, such as extra resource consumption and active-message overhead, are extremely low comparing to capacity of the various computing platforms, including mobile devices that are used as vehicle GPS devices, as validated by our experiments.

1.1 Terminologies

The following terminologies will be used in this paper:

- *Synchronization* is the process of coordinating the behaviors of collaborative programs in a distributed application. When the behavior of a local program is reconfigured to adapt to changing contexts, it requires its peer programs to change their behaviors correspondingly for system consistency.
- *Synchronous synchronization* means the synchronization is realized through a synchronous method that requires all synchronization participants to complete changing their behaviors at the same time and suspend their application-layer operations in this process.
- *Asynchronous synchronization* means the synchronization is realized through an asynchronous method, in which the local program can resume its operation right after its own behavior is changed for adaptation and other synchronization participants reactively change their behaviors only when they communicate with this local program.
- *Sensor* is the hierarchical context events that can measure and evaluate specified contexts at runtime and notify subscribed actuators for adaptation.
- *Actuator* is a reflective component that contains a set of functional components to form a functional path or component chain, which process application-layer data, and a meta-interface, which can represent its internal states and reconfigure the component properties or chain structure of the actuator at runtime.
- *Active actuator* means the actuator status is active. There is one and only one actuator active at any time and only the component chain in the active actuator processes application-layer data. Various actuators can be activated or deactivated to adapt to changing context according to user-defined policies.
- *Proactive actuator* is constructed at the system initialization phase to process local data and proactively change its behaviors to adapt to changing contexts at runtime according to user-defined adaptation rules.
- *Reactive actuator* is constructed at the system synchronization phase to process received data from peer programs and reactively change its behaviors according to the active message header of the received data packet.

This rest of the paper is organized as follows. Section 2 covers the related work. Section 3 describes the details of the MARCHES architecture. In Section 4, we have implemented and validated MARCHES in our experiments. Finally, we conclude this paper and present some future work in Section 5.

2. Related Work

2.1 Advances and Applications in Vehicular Ad Hoc Networks

The research and development of vehicle safety communication (VSC) techniques and their supported vehicle applications are becoming more and more popular worldwide for their advantages to collision/violation warning and avoidance in vehicle systems. In US, the Vehicle Infrastructure Integration (VII) [1] project, which aims to create a nationwide enabling communication infrastructure by connecting vehicles and roadside infrastructures, has been initiated by the US Department of Transportation (DOT) and supported by many state DOTs and auto companies. VII contains three major function modules. On-board equipments are used to collect situational contexts, like the information of local and nearby vehicles, and provide driver interfaces; Road-side equipments can communicate with on-board equipments to exchange information based on DSRC protocols; and regional message switches send the information to remote end users that can process the data and send back control commands. The research and development of VII focuses on the research of DSRC standards, like IEEE 1609- 1, 2, 3, and 4, and development of vehicle and roadside equipments and applications, like signal violation warning and stop sign violation warning etc.

In Japan, the National Police Agency is promoting the Next Generation Universal Traffic Management Systems (UTMS21) [4], which aim to create an environment-friendly and traffic-oriented society to smooth traffic flow and avoid traffic accident.

It integrates eight subsystems for vehicle context gathering and management, intelligent vehicle control, responsibility to emergent situations. One of the most important subsystems in UTMS21 is Driving Safety Support System (DSSS) that not only studies and classifies the causes and solutions of traffic accidents, but also utilizes the latest techniques for context collection and collision prevention, like the danger zone avoidance control system, the unforeseeable collision warning system, the right-turning vehicle-caused accident prevention system, and the pedestrian crossing support system. Advanced Cruise-Assist Highway Systems (AHS) [3] is another advanced VSC systems in the ITS field, which aims to reduce traffic accidents and congestion and reduce the operational work of drivers as well. AHS contains three themes. AHS-"i" (information) focuses on gathering contextual information like other vehicles, obstacles, and highway surface condition etc; AHS-"c" (control) focuses on vehicle control assistance; and AHS-"a" (automated cruise) focuses on providing fully automated driving.

In Europe, the integrated project PReVENT [6] for preventive and active safety applications has been proposed to help drivers avoid critical situations in advance, or avoid accidents in the critical situations, or reduce the severity of accidents if they are not avoidable. There have been many advanced sensing, digital map and positioning, and wireless communication techniques developed and further integrated in dedicated demonstrator platforms for safety applications. The activities in PReVENT can be classified into vertical fields that target on the independent development of a single safety function, like the speed sensing function and control function, and horizontal fields that target on the interaction and integration of these functions to develop intelligent vehicle applications. Another VSC organization initiated by European vehicle manufacturers is C2CCC (the CAR 2 CAR Communication Consortium) [5], which is dedicated to further increase of road traffic safety and efficiency. Different from the projects in US and Japan that target on DSRC techniques, C2CCC uses the standard IEEE 802.11 and Wireless ad-hoc network techniques for inter-vehicle communication and vehicle to roadside infrastructure communication. A list of active safety applications based on the car 2 car communications have been proposed to provide advanced driver assistance, user communications and information services.

This work does not focus on the research of communication techniques or development of special vehicle applications. Instead, we introduce a middleware framework to fill the gap between them and improve the application affordability, flexibility and adaptability while satisfying their critical real-time requirements.

2.2 Middleware techniques

Middleware has been a critical technology for developing distributed applications because it can mask the heterogeneity of the underlying environment and provide an integrated service environment to simplify the task of programming and managing applications. It can be further separated into the multiple layers (shown in Fig. 3) to provide various functions for vehicle systems.

Communication middleware focuses on integrating distributed computing systems to act as a unified resource to reduce the application development cost. Early stage middleware, like the Common Object Request Broker Architecture (CORBA) [10], the Distributed Component Object Model (DCOM) [11], and Java Remote Method Invocation (RMI) [12], builds on Remote Procedure Call (RPC) to abstract the low-level TCP/IP communication details and replace the communication interface with a local procedure call or function invocation. Unlike RPC-based middleware, Message Oriented Middleware (MOM) [15, 17], provides an asynchronous communication mechanism for distributed applications based on message exchanges. MOM improves the system flexibility and robustness as the change of one client does not require the change of other clients (called loose coupling). Further, the asynchronous communication improves the system efficiency by allowing the processing parallelism, in which the communication caller can continue processing regardless of the state of the messages and peer agents.

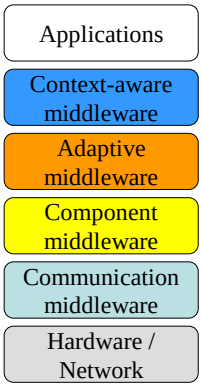


Figure 3. Middleware layers

Component middleware, normally based on a component model (e.g. CORBA Component Model (CCM) [35]), enables reusable service components to be organized, configured, and deployed to develop applications efficiently and robustly. A component is a service entity that exposes a set of interfaces, which components use to communicate with each other for collaboration, and attributes, which specify its parameters that can also be reconfigured at run-time via component metadata. Component middleware provides standards for object implementation and interaction so that it can support generic service components and then reduces the complexity of software upgrades and increases the reusability and flexibility of vehicle applications. Existing component middleware techniques contain both reusable common services, e.g. optimization of resource consumption (OSA+ [36], ACE [37]), configurability, (Zen [38], TAO [39]), and reusability (nORB [40]) etc, and domain-specific services, e.g. OSEK/VDX [41] for vehicle applications, and ARINC 653 for avionics.

Adaptive and reflective middleware [42, 43] can inspect its internal representation at runtime and reconfigure its state and behavior by providing a set of meta-level interface or object for base-level implementation that handles the real service execution and operations. Open ORB [44] provides both structural reflection and behavioral reflection. The structural reflection supports functional components reconfiguration. It has two meta-models: the interface meta-model, which allows dynamically recover a component's interfaces at run-time, and the architecture meta-model, which provides access to the component architecture (Open ORB components are organized in a hierarchical way and a component may contains multiple sub-components). The behavioral reflection supports nonfunctional components reconfiguration. It enables the dynamic insertion of interceptors on a specific interface to introduce nonfunctional behaviors into the ORB, such as security checks and concurrency control etc. Dynamic TAO [45] is a reflective ORB based on a collection of component configurator. The domain configurator maintains the references of a TAOConfigurator and a set of servant

configurators. The TAOConfigurator can then inspect and dynamically change nonfunctional behaviors of the middleware.

Context-aware reflective middleware actively measures the application interested contexts and adapts to them automatically and predicatively to meet the adaptability demands of vehicle applications. The classification and measurement of the context, which includes network awareness, device awareness, user awareness, application awareness, and environment awareness, has been studied in our previous work [31]. QuO [46] can monitor the application status at runtime and dynamically allocate QoS resources robustly based on the adaptive and reflective model to accommodate the rapidly changing environmental conditions and application requirements. However, similar with the Dynamic TAO, QuO does not consider the synchronization delay in the reconfiguration process while supposing that the reconfiguration has been constrained in safe conditions in advance.

MobiPADS [13] is a client server based context-aware middleware that supports both middleware-layer (nonfunctional behaviors) and application-layer (functional behaviors) adaptation for mobile devices. It takes advantage of a communication channel for synchronization in a synchronous way each time the architecture is reconfigured. The reconfiguration process includes operation suspension, buffer clearance, and chain-structure modifications. Because the initializer of the synchronization has to be suspended until the system architecture of its own and other participants is reconfigured and the buffered data for previous architecture is cleared, the reconfiguration time is in a range of seconds or even more according to their experiments. CARISMA [14] employs a novel micro-economic approach that relies on a particular type of sealed-bid auction to handle the conflicts. The conflict resolution algorithm includes run-time conflict detection, solution set computation, and bids computation processes for each reconfiguration. However, the approach is still synchronous and inherits all the above disadvantages. Different from the existing context-aware reflective middleware frameworks that use the synchronous synchronization, MARCHES maintains multiple component chains and leverages the active message technique to realize the synchronization in an asynchronous way. According to the discussion in Section 3 and evaluation in Section 4, MARCHES can significantly reduce the reconfiguration time compared to existing context-aware reflective middleware.

2.3 Active messages

The concept of active messages was originally proposed for large-scale multiprocessors to minimize inter-processor communication overhead and allow communication to overlap computation [26]. This concept has then been widely used in parallel and distributed computing systems to reduce communication overhead [27, 28]. Recently, it has been used in wireless sensor network research to avoid busy-waiting for data to arrive and overlap communication with other sensor activities [29]. This paper utilizes the active-message concept for the first time to address the behavior synchronization problem of the vehicle middleware.

3. System Architecture of MARCHES

As shown in Fig. 4, MARCHES is located between the lower hardware-and-network layer and the upper application layer to monitor environments and support vehicle application adaptation. It is peer-to-peer middleware with one middleware agent per

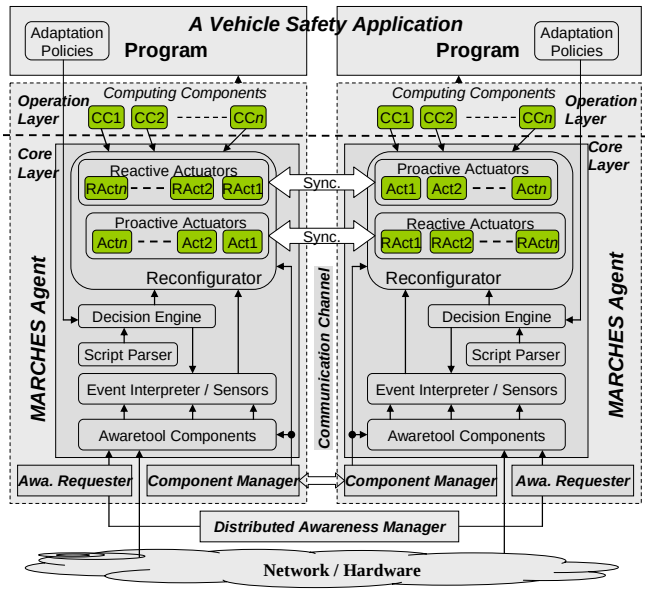


Figure 4. System architecture of MARCHES.

application in each host. Each MARCHES agent can be separated into a core layer and an operation layer. The core layer consists of eight function modules, which construct the adaptive and reflective framework for monitoring contexts and reconfiguring the system behaviors. Measurement tools and a hierarchical event interpreter deal with context measurements and evaluations by building hierarchical context events (sensors); a decision engine, a dynamic reconfigurator, and an XML-based script parser support the MARCHES reflection model and efficient reconfiguration; in addition, a communication channel for the inter-communication among peer agents, a distributed awareness manager for disseminating and synchronizing awareness results [31], and a component manager for supporting and managing MARECHES components are designed as surrounding services.

The measurement tools are the lowest building blocks that monitor the dynamic environments and report the awareness results as the contextual information to be processed by the event sensors. The sensors and actuators, in addition to adaptation policies, are defined by application developers or users in a XML script file. There are two types of actuators: proactive and reactive ones, each of which contains a component chain and performs reconfiguration actions. The XML script parser parses the script file and constructs the sensors and proactive actuators to process local data. The reactive actuators are constructed through the synchronization process with peer agents to process the received data. Once a context triggers an event sensor, a corresponding proactive actuator will be activated by the reconfigurator to perform the reconfiguration actions.

In the operation layer, various services are offered by software components that implement specific algorithms and protocols. There are two types of components in the operation layer: functional components (called *marchlets*) for performing vehicle communication and control, and context-awareness components (called *marchtools*) for measuring and evaluating situational contexts. Because we focus on improving the vehicle middleware efficiency in this research, we do not include such nonfunctional

components as the concurrency and security etc. and their reconfiguration, which are potentially supported by MARCHES.

3.1 MARCHES Reflective Model

MARCHES supports both component-level and system-level reflection. The component-level reflection deals with the content and behavior of a given component via the interface metamodel, which provides discovery of and access to the set of provided and required interfaces of the component. Based on the component-level reflection, MARCHES supports standard reflective software components or other third party components in a cost-efficient manner, so that it is easily upgradeable to incorporate new techniques or services in its operation layer and meet the rapid progress of new algorithms and standards for vehicle applications. The system-level reflection deals with the structure and graph of the component connections via the architecture metamodel, which provides discovery of and operation to the current active actuator. The system-level reflection allows MARCHES to examine its internal states at run-time and dynamically reconfigure the application architecture to enhance its adaptability.

3.1.1 MARCHES components and component-level reflection

A MARCHES component is a function independent reflective element that provides an interface metaobject by which a component can read its own metadata, extract the metadata from the component (called reification), and use that metadata either to inform the component user or to modify the component's behavior (called absorption). Metadata is information about the data—that is, information about the types, functions, code, and etc., which are stored along with a component. By using the interface metamodel and component-level reflection, MARCHES can examine the types in a component, create new types at runtime, interact with or instantiate the types, and dynamically invoke properties and methods on the instantiated objects [30] (called the *late binding*).

To incorporate a new reflective component in MARCHES, users need to describe the types, interfaces, and other attributes of the component in a system script file by using our defined IDL (Interface Description Language), as shown in Fig. 5, so that the component can be identified and configured by MARCHES at runtime through the late binding. We have realized three methods to identify a MARCHES component for vehicle applications: the exclusive component *name* for a registered system component, the complete *address* for a local component, and the desired attributes for a registered component in the component manager. The component type is declared in the *ctype* part and the *alias* is the name of the component used in the adaptation-rule part of the script. The component can be specified by setting its parameters, which can also be reconfigured at runtime according to the adaptation rules. It also provides some *interfaces* (e.g. input and output interfaces). The connected input and output interfaces must support compatible event messages and their connections can also be reconfigured at runtime.

There are two types of MARCHES components: reconfigurable functional components (*marchlets*) and extensible context-awareness components (*marchtools*). The context-awareness components can be further classified into measurement tool components (named as *awaretools*) and user defined function components (named as *awarefuncs*) built above awaretools.

```
<component cid="2002">
 <addr> D:\Masslets\JPEGCompress.dll </addr>
 <name> Masslets.Compress.JPEGCompress </name>
 <ctype> Masslet </ctype>
 <alias> COMPRESS </alias>
 <param pid="001">
  <name> SetCompressQuality </name>
  <vtype> Int32 </vtype>
  <value> 50 </value>
 </param>
 <interface iid="001">
  <name> PtrDataInput </name>
  <itype> Input </name>
  <Message> PDIBEventArgs </Message>
 </interface>
 <interface iid="002">
  <name> DataOutput </name>
  <itype> Output </name>
  <Message> JPEGEventArgs </Message>
 </interface>
</component>
```

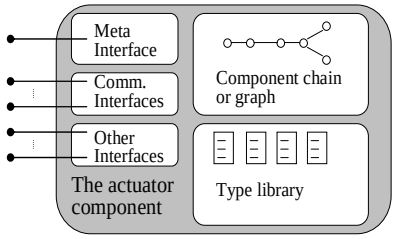
Figure 5. The component declaration in MARCHES

Marchlets are the basic functional units to construct vehicle applications. MARCHES supports the publish/subscribe model for communication and each marchlet provides some output and input interfaces for component assembly. The output interface of a marchlet can be subscribed by the message-compatible input interfaces of other marchlets and publish messages to them.

Measurement tools, which measure and predict real-time context awareness e.g. network conditions for vehicle communication and road situations for action replan in MARCHES, are realized as reflective components (called awaretools) to facilitate the reuse and extension of existing measurement tools. The context awareness research has been addressed in our previous work [31]. Awaretools act as the lowest event sources in the event tree that can be subscribed by higher level event nodes and organized in a hierarchical way to build event sensors.

MARCHES also supports another type of reflective component—awarefunc to assist the extension of awaretools. An awarefunc provides not only output interfaces to accept the subscription from, and notify higher level sensor nodes, but also some input interfaces to subscribe to awaretools and process their raw data as input parameters. Thus it supports user defined functions for pre-processing the measurement results of awaretools, e.g. getting the average value of the bandwidth in the last 5 minutes.

To better maintain and update MARCHES components, we have proposed a distributed service module, called the Component Manager (CM) that accepts component registration and provides such services to vehicle applications as component identification, evaluation, migration, and virtual connection. So far we have focused our concern on the component employment and realized component identification function according to a user-provided component name and version properties; thus we leave the component evaluation [32], migration [33], and virtual connection [34] functions for further work.



(a)

```
interface IMetaActuator
{
   componentList get_components();
   connectionList get_connections();
   bool set_components(componentList compList);
   bool set_connections(connectionList connList);

   bool add_component(CMarchletObj marchlet);
   bool remove_component(string marchlet);

   bool connect_all_components();
   bool disconnect_all_components();

   bool connect_components(string senderObj,  string senderInterf,
      string receiverObj,  string receiverInterf);

   bool disconnect_components(string senderObj,  string senderInterf,
      string receiverObj,  string receiverInterf);

   Object get_component_parameter(string comp,  string param);
   bool set_component_parameter(string comp, string param, Object value);

   EnumActuatorStatus get_active_status();
   bool activate();
   bool deactivate();
   ……
}
```

(b)

Figure 6. (a) The MARCHES actuator architecture and (b) its meta-interface

3.1.2 MARCHES Reconfigurator and System-level Reflection

MARHES reconfigurator contains multiple actuators and provides interfaces to manipulate the actuators so that the application behaviors can be reconfigured. The actuators are designed as reflective components to support the MARCHES system-level reflection. Each actuator (see Fig 6 (a)) contains a component chain (or graph) for processing application data, a type library for browsing the component types, and a meta-interface presented in Fig 6 (b) for its C# code. The meta-interface provides access to its underlying meta-information and internal states (reification), including the structure and graph of the component connections and the actuator status (active/inactive) etc. By accessing the meta-interface, the reconfigurator can change the actuator meta-information that leads to a change of the actuator implementation (absorption), including the structure modification of component connections and the switch of active and inactive actuators.

3.2 MARCHES Efficient Reconfiguration

For a distributed adaptive system, a reconfiguration process consists of two steps: the local behavior change that is triggered actively by the context changes and the distributed behavior synchronization that is triggered reactively to synchronize local behaviors with the changed behaviors of other distributed programs for safe reconfiguration.

3.2.1 Local behavior reconfiguration

In traditional reflective middleware, there is only one component chain (or graph) in each middleware agent. The reconfiguration process is then to modify the chain (or graph) structure. However, due to the operation suspension and synchronous synchronization during the reconfiguration process, both the overhead and delay are too large to be tolerable for real-time vehicle applications.

By contrast, MARCHES supports multiple component chains as shown in Fig. 2, each of which is located in an actuator. Every actuator is hooked to an event sensor that is described in the same policy. There is one and only one actuator active at any time and only the component chain in the active actuator processes the application data. To reduce the resource consumption by multiple chains, each actuator only maintains a chain of references, which point to marchlet instances, and a customized parameter list for each reference. When contexts change and satisfy a new sensor, the sensor will notify the decision engine for reconfiguration by switching active and inactive actuators. That is, the current active actuator is deactivated by suspending its operations as it reaches a safe state, where the reconfiguration can be safely performed, storing its run-time status, and disconnecting the component chain; the target actuator is then activated by connecting its components, restoring the run-time status, and resuming its operations.

3.2.2 Distributed behavior synchronization

Because each distributed middleware agent has its own executive component chain in the active actuator, behavior synchronization is a crucial service of the middleware to achieve behavior consistency for vehicle applications in the reconfigurations. We have designed and implemented an efficient asynchronous synchronization protocol in MARCHES based on active messages (the terminologies of synchronous and asynchronous is borrowed from the synchronous and asynchronous communication model in RPC-based middleware and MOM depicted in the related work section). The basic idea of the proposed asynchronous protocol is that every middleware agent constructs the reactive actuators for all connected agents in the initialization phase, and selects one of them to process each received application layer data packet according to the active message header attached in the packet. The synchronization protocol has five initialization steps, such as those shown in Fig. 7.

- In the middleware initialization phase, proactive actuators of each agent are constructed based on a user-defined script file. Each proactive actuator is also associated with a middleware-assigned unique index and the architecture-information of an optional reactive actuator.
- The middleware agent then sends a *synchronization request* packet to each distributed peer agent. The packet contains the indices of the proactive actuators and the architecture-information of the reactive actuators.
- After receiving the *synchronization request* packet, the peer agent constructs the reactive actuators according to the packet information, each of which is associated with the IP address of the packet sender and a middleware-assigned unique index as well.

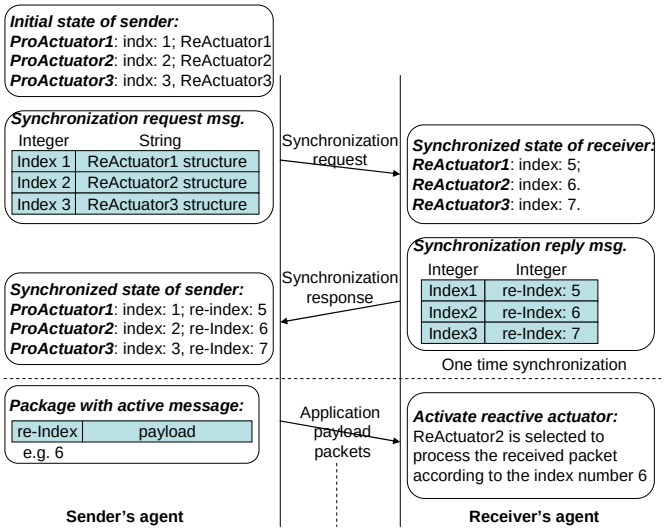


Figure 7. The synchronization process in MARCHES

- The receiver or the peer agent replys the sender a *synchronization response* packet that contains a set of index pairs, each of which contains an index of the proactive actuator and the index of the reactive actuator.
- The sender agent replaces the architecture-information of each reactive actuator with its corresponding index received from the synchronization response packet.

The above-mentioned initialization is a one-time process for each peer agent. Then the middleware agent appends the index of the reactive actuator, which corresponds to the current active actuator, to the payload of each data packet. The peer agent receiving the data packet activates the reactive actuator indexed by the received index to process the data packet correctly.

The active message based asynchronous synchronization protocol has four advantages: low overhead, short delay, high efficiency, and better robustness. In general, only the index of the reactive actuator needs to be stored in the active message header for each data packet. By using the asynchronous method, the system does not need to be paused in the synchronization process, which dramatically reduces the reconfiguration time. Further, based on the information in the active message header, a peer agent can always process the received packets by choosing the correct reactive actuator and then no buffered clearance is needed in the reconfiguration process, which makes the reconfiguration by our middleware efficient. Moreover, once the reactive actuators are constructed in the system initialization phase, the local agent reconfiguration does not require the availability of other agents and then is not affected by the network condition or the capacity of other agents. Thus the robustness of the application is improved and the communication overhead is reduced.

MARCHES is rule-based reflective middleware, in which all adaptation rules (or policies) are predefined and applications then adapt to contexts according to the rules. However, it may be difficult to predict all the desired policies in advance and users may need to change the policies based on the vehicle runtime status. MARCHES provides a re-synchronization method to support the runtime modification of adaption policies. That is, the agent suspends its operations, clears data buffer, re-synchronizes the modified policies with peer agents, and resumes its operations.

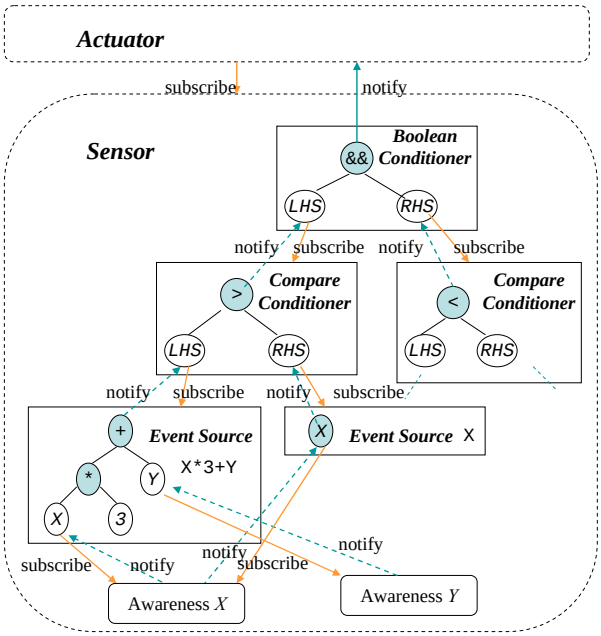


Figure 8. The binary tree based event notification model.

3.3 MARCHES Event Model

Situational contexts are very important information in vehicle applications for either drivers or autonomous control systems to avoid traffic accidents, especially in complicated environments like cross streets and crowd highway lanes etc. Existing context-aware middleware only supports simple context monitoring or the combination of simple contexts. To improve the expressive ability and comprehensiveness for context evaluation, a binary tree based hierarchical event notification model is proposed (see Fig. 8), in which several events can be organized and integrated as a binary tree structure to construct an event sensor to monitor and process interested contexts and trigger subscribed actuators at runtime for reconfiguration when the conditions of the sensor are satisfied. MARCHES event model supports both context expressions and user-defined functions by applying *awarefunc* components. To enhance the efficiency of system reconfiguration, MARCHES contains multiple event sensors for supporting multiple actuators, so that each actuator can subscribe to dedicated event sensor.

In the constructed event tree, each event node contains a *conditioner*, a left hand side (*LHS*), and a right hand side (*RHS*). There are two types of conditioners: the *compare conditioner* and the *Boolean conditioner* that do compare and Boolean operations on LHS and RHS respectively. The LHS and RHS can subscribe to the conditioner of a lower-layer event node or an event source, and notify the conditioner when the lower-layer contexts or operation results are changed. The event source can be a constant value, single context awareness, or an awareness expression that is also built on a binary tree structure, in which each node has an operator, a LHS, and a RHS. Therefore, all the contexts are organized in a hierarchical way to form a sensor. The event nodes in sensors communicate based on the publish/subscribe model. An upper-layer conditioner or an actuator can subscribe to an event

```
<sensor>
 <event>
  <otype> And </otype>

  <lhs>
   <event>
    <otype> GT </otype>
    <lhs>
     <expr> Ave(AVI_BW, 5) </expr>
    </lhs>
    <rhs>
     <expr> 10 </expr>
    </rhs>
   </event>
  </lhs>

  <rhs>
   <event>
    <otype> LT </otype>
    <lhs>
     <expr> Ave(AVI_BW, 5) </expr>
    </lhs>
    <rhs>
     <expr> 20 </expr>
    </rhs>
   </event>
  </rhs>
 </event>
<sensor>
```

(a)

Script Operator	Development Operator
GT	>
GE	>=
LT	<
LE	<=
NE	<>
EQ	==

(b)

```
Ave(AVI_BW, 5)>10 &&
Ave(AVI_BW, 5)<20
```

(c)

Figure 9. A sensor declaration example in the XML script file: (a) XML-based sensor script example, (b) Script operator to development operator mapping, (c) user development example

node or a sensor as a listener. Thus the sensor can monitor and process the awareness results according to the event tree, and eventually report interested events to the subscribed actuator and trigger the application reconfiguration.

To improve the efficiency of sensors, the hierarchical event tree is constructed according to the Modified Directed Acyclic Graph (MDAG). That is, before creating a new event node, check whether an identical node or an inverse node already exists. Event node *a* is defined as the inverse node of *b* if a and *b* have the same event source and comparison value, but inverse comparison operator, e.g. the inverse event of "Min(AVI_CPU, 10) < 1.0" is "Min(AVI_CPU, 10) 1.0".

For various application developers or end users, to use the event model to identify their interested contexts, they can declare the sensors they want to specify in the script file as shown in Fig. 9 (a). The example means when the average bandwidth during the last 5 seconds is greater than 10Mbps and less than 20Mbps, the sensor notifies its subscribed actuators. To facilitate the configuration of the sensor script, we developed a tool, which is part of the script development tool that is discussed in section 3.4, to transfer the advanced language to the script language according to the operator mapping (Fig. 9 (b)). Fig. 9 (c) is the advanced language example of Fig. 9 (a)

3.4 MARCHES Description Language and XML Script file

To facilitate the development of MARCHES supported vehicle applications, an XML-based system script file is provided for application developers or end users to customize the application configuration and adaptation policies. In particular, the script file can be divided into a declaration part and an adaptation-rule part (see Fig. 10). The declaration part declares all masslets and masstools – components used in a local program and middleware agent. The detail of a component declaration is shown in Fig. 5. Based on the declaration, the MARCHES agent loads and

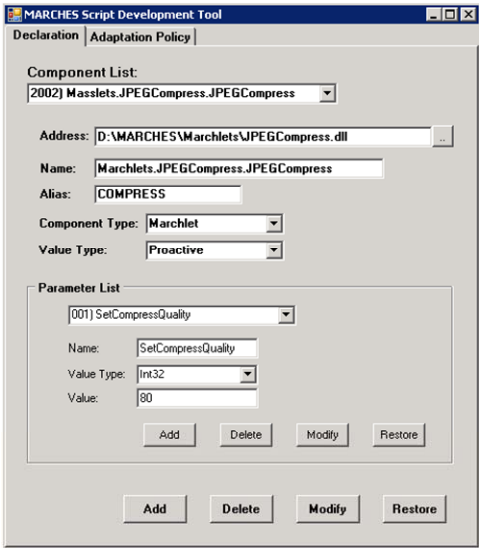
```
<Masslets>
  ...
</Masslets>

<MassTools>
  ...
</MassTools>

<Rules>
 <rule>
  <sensor>
   ...
  <sensor>

  <Actuator type="proactive" sync="Async">
   <SetParam>
    COMPRESS.CompressQuality = 70;
   </SetParam>
   <SetArch>
    Grab.PtrDataOutput -> COMPRESS.PtrDataInput;
    COMPRESS.DataOutput -> Socket;
    Grab.Start;
   </SetArch>
  </Actuator>

  <Actuator type="reactive" sync="Async">
   <SetArch>
    Socket -> DECOMPRESS.DataInput;
    DECOMPRESS.DataOutput -> DISPLAY.DataInput;
   </SetArch>
  </Actuator>
 </rule>
 ...
</Rules>
```

Figure 10. An example of the adaptation rule script.

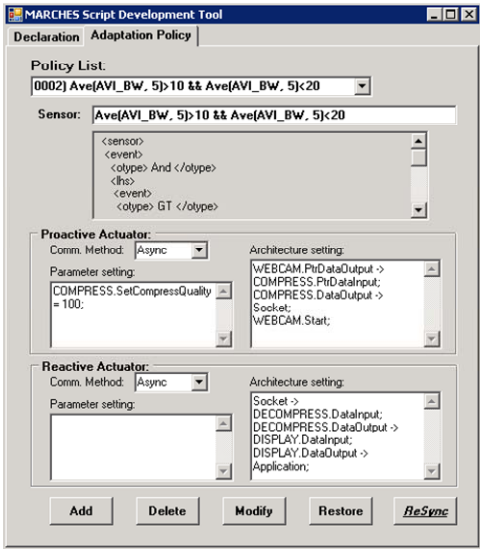
instantiates the components by the component-level reflection, and initializes them with the provided parameters.

The adaptation-rule part contains adaptation policies and each policy can be further separated into three sections: a sensor, a proactive actuator, and an optional reactive actuator. The sensor section can be parsed by the event interpreter to build an event sensor that monitors interested contexts and accepts the subscription of the proactive actuator declared in the proactive actuator section. The proactive actuator section describes the system architecture-information that is used to update the actuator internal states by the reconfigurator when it performs reconfiguration actions. Therefore, the system behaviors can be dynamically adapted to the context changes through the MARCHES system-level and component-level reflection (respectively, architecture reconfiguration and parameter tuning). The reactive actuator section describes the architecture-information of an actuator in peer agents that processes the received data from the proactive actuator, so that the behaviors of the proactive and reactive actuators can be synchronized in distributed vehicle systems.

For the image transmission example discussed above, the component chains in proactive actuators prepare and send video frames and the chains in reactive actuators receive and display the frames. There are four marchlets (*Grab*, *Compress*, *Decompress*, and *Display* components), two awaretools (available bandwidth and CPU measurement tools), and two awarefuns (average and minimum functions). The application behavior can be dynamically reconfigured by using or not using the Compress marchlet according to the two adaptation polices, e.g. the first policy means that when the average available bandwidth in the last 5 seconds is less than 10Mbps, the sensor notifies the reconfigurator to active the proactive-actuator that connects the Grab and the Compress components for reconfiguration.



(a) Component declaration



(b) Adaptation policies

Figure 11. The MARCHES script file development tool

To facilitate interactive configuration, we developed an XML-script Development Tool. As shown in Fig. 11, the tool is a C# graphical user interface that lets users manipulate both the component and policy configuration and runtime reconfiguration interactively. The tool supports the advanced language to describe event sensors and establishes a connection to the MARCHES middleware system to help the users re-synchronize the local agent with peer agents by using the mouse when the adaptation policies are reconfigured at run-time.

4. Implementation and Performance Evaluation

MARCHES aims at improving the reconfiguration efficiency of traditional context-aware reflective middleware and supporting real-time vehicle applications in dynamic environments by using active messages to coordinate reconfigured application behaviors asynchronously. However, the reconfiguration process introduces some performance cost such as extra resource consumptions to maintain the multiple chains. Therefore it is important to check the feasibility and efficiency of using MARCHES. We have analyzed the robustness of MARCHES in the above sections, and we evaluate its performance cost and benefits in this section in terms of the memory footprint, reconfiguration time, and scalability through benchmark applications.

In our experiments, we have implemented the MARCHES system in C# for both the Windows XP system (WXP) and the Windows Mobile 5 system (WM5) using visual studio 2005 and encoded the system script file using XML (eXtensible Markup Language). The benchmark and application testbed consists of two laptops (Thinkpad-X60: Intel T2300 1.66GHz, 512MB PC2-5300, and Windows XP), two PDAs (Dell x51v: Intel XScale 624MHz, 64MB, and WM5) and six desktops (Dell dimension 4550: Pentium 4 2.66GHz, 512MB) to simulate vehicle and roadside environments. The PDA hardware configuration is compatible with that of most on-board GPS devices and the laptop and desktop configuration is similar with that of the roadside and remote base station devices. We test the memory footprint, reconfiguration time in a local setting and evaluate the system scalability in a distributed setting on both PCs and PDAs.

4.1 Memory Footprint and Active Message Overhead

In this experiment, we evaluate the local storage size and the run-time memory consumptions of MARCHES system, components and actuators. We utilize the C# serialization function to serialize MARCHES and system objects and measure their run-time memory usage, like the socket and XML-parser objects. Serialization means that objects are marshaled by value, that is, all their various member data are written out to the stream as a series of bytes. Therefore, we can use the length of the stream as the memory consumption.

Table 1. MARCHES middleware and resource consumption.

Components	Windows XP system		Windows Mobile 5	
	Local file size	Run-time memory usage	Local file size	Run-time memory usage
Middleware (MARCHES.dll)	56KB	896KB	46KB	123KB
Empty Masslet (eMasslet.dll)	4KB	139Byte	4KB	74Byte
Simple Masslet (sMasslet.dll)	16KB	356Byte	5KB	147Byte
Simple Masstool (sMassTool.dll)	16KB	279Byte	4KB	94Byte

The local file size and run-time memory usage of MARCHES middleware and components (marchlets or awaretools) are shown in table 1 for both WXP and WM5 respectively. The run-time memory usage of the middleware is measured after the system initialization and before loading and instantiating any components. The empty masslet is an empty reflect component without containing any application-specific methods and variables

and the simple masslet containing one input interface, one output interface, and 5 double-type parameters. Although we use very similar source code for both WXP and WM5, the run-time memory consumption is much different due to the mobile platform code optimization.

Table 2. Parameter notation of resource consumption.

Notation	Parameter	Default
p_{ijk}	The size of parameter *k* for marchlet *j* in acutator *i*	10 bytes
l_{ij}	The reference and name size of marchlet *j* in actuator *i*	12 bytes
a_i	The index size of actuator *i*	8 bytes

Because MARCHES contains multiple actuators, each of which consists of a complete chain of components, it is important to analyze the overhead of the multiple actuators. Table 2 describes the notation used in the analysis of resource consumption for maintaining multiple actuators. The memory consumption *R* is then expressed as:

$$R = \sum_i \left(\sum_j \left(\sum_k p_{ijk} + l_{ij} \right) + a_i \right) . \quad \textbf{(1)}$$

For the MARCHES agent in our example that contains 2 actuators as described in Fig. 10, the resource consumption is 102 bytes. For a more complicated MARCHES agent that contains 5 actuators, 10 marchlets for each actuator, and 10 parameters for each marchlets, the resource consumption is 5640 bytes (≈ 5.5 KB) that is still fairly small for most on-board vehicle devices hosting megabits or gigabits memories.

The overhead of the one-time synchronization initialization process includes the *synchronization request* (architecture-information of the reactive actuators) and *synchronization response* (index pairs) packets that are much smaller than the payload size of a data packet. After the initialization, only an active message header is appended to each data packet to store the index of a reactive actuator.

4.2 Time Efficiency (middleware performance in a local setting)

The major goal of MARCHES is to improve middleware efficiency. In this section, we evaluate comprehensive time efficiency of MARCHES system according to some benchmarking applications, including: component loading time, reconfiguration time, event sensor response time, and message exchange time.

Table 3. Benchmarking MARCHES component loading time.

	Comp. #	1	10	100	1000	10000
PC	Ave. (*ms*)	0.163553	0.863828	8.950579	83.71129	875.7631
	Standard Dev. (*ms*)	0.003973	0.015096	0.188331	1.091953	2.016096
	Confidence Int. (α=0.5)	0.000893	0.003394	0.042342	0.245504	0.453279
PDA	Ave. (*ms*)	1.693265	190.6756	2211.647	24387.93	
	Standard Dev. (*ms*)	0.024551	0.382868	2.097835	61.17155	
	Confidence Int. (α=0.5)	0.00552	0.08608	0.471656	13.75319	

4.2.1 Component loading time

The component loading time is defined as the time period of loading a masslet, checking its type, and instantiating the object based on the parameters encoded in the script file. The maximum number of component files stored in a PDA folder is 1000. All the data are calculated based on 10 repeated tests, as shown in table 3.

4.2.2 Behavior reconfiguration time

By using active messages, MARCHES reconfiguration is realized by switching component chains, which disconnects the current component chain in the active actuator and connects the component chain of the triggered actuator by an event sensor. Thus we have:

$$. T_{rec} = T_{disc} + T_{conn} \quad \textbf{(2)}$$

The component connection time and disconnection time in our experiment are shown in table 4 and 5.

Table 4. Benchmarking MARCHES component connc. time.

	Comp. #	1	10	100	1000	10000
PC	Ave. (*ms*)	0.039887	0.351814	4.413969	40.97256	415.5276
	Standard Dev. (*ms*)	0.000271	0.023946	0.771114	0.658318	6.238717
	Confidence Int. (α=0.5)	6.1E-05	0.005384	0.173369	0.14801	1.40265
PDA	Ave. (*ms*)	0.564239	3.717504	41.76759	411.3645	
	Standard Dev. (*ms*)	0.006986	0.025047	1.066252	25.64663	
	Confidence Int. (α=0.5)	0.001571	0.005631	0.239725	5.766131	

Table 5. Benchmarking MARCHES component discon. time.

	Comp. #	1	10	100	1000	10000
PC	Ave. (*ms*)	0.041129	0.361033	4.450224	41.90418	421.084
	Standard Dev. (*ms*)	0.001974	0.029886	1.028386	0.656612	3.402015
	Confidence Int. (α=0.5)	0.000444	0.006719	0.231212	0.147626	0.764875
PDA	Ave. (*ms*)	0.533162	3.836	44.56978	448.623	
	Standard Dev. (*ms*)	0.009623	0.018377	2.366826	34.49798	
	Confidence Int. (α=0.5)	0.002164	0.004132	0.532133	7.756178	

The experimental results demonstrate that the architecture reconfiguration time is in the range of hundreds of microseconds when the middleware needs to change 10 components for reconfiguration, and is several milliseconds for changes of 10,000 component connections. Further, the reconfiguration time in

MARCHES is only determined by local hardware resources so that the time is very stable. By contrast, the reconfiguration time of traditional middleware systems is highly related with network conditions as they use synchronous synchronization protocols. For example, MobiPADS takes 10*s* for 5 component deletion and 5 component addition operations when the bandwidth is 20kbit/s and 3*s* for the same operations when bandwidth is 1Mbits/s according to their experimental results [13].

4.2.3 Event sensor notification time

Event sensor notification time is another important metric to evaluate the system responsiveness to environments. To evaluate the infection of the event structure to the notification time, we test the time based on a linear tree mode, where there is only one event source in each level, and a binary tree mode, where different awaretools are constructed to a full binary tree. Therefore, the number of simple event sources (awaretools) used in the linear tree mode is a constant 1, and the low level composite event is used as the high level event source. The number of simple event source used in the binary tree mode is related with the depth of the tree, and we have $N_{tools} = 2^n$ where n is the event-tree level. The test results are shown in table 6 and 7.

Table 6. Benchmarking linear tree event notification time

	Level #	2	5	10	15	100
PC	Ave. (*ms*)	0.00596	0.006115	0.006373	0.006674	0.01164
	Standard Dev. (*ms*)	0.000279	0.000168	0.000286	0.000381	0.00152
	Confidence Int. (α=0.5)	6.28E-05	3.77E-05	6.42E-05	8.57E-05	0.00034
PDA	Ave. (*ms*)	0.104376	0.09494	0.109778	0.116957	0.51374
	Standard Dev. (*ms*)	0.000894	0.004208	0.005677	0.006546	0.00863
	Confidence Int. (α=0.5)	0.000201	0.000946	0.001276	0.001472	0.00194

Table 7. Benchmarking binary tree event notification time

	Level #	2	5	10	15
PC	Ave. (*ms*)	0.00590	0.00621	0.006736	0.01288
	Standard Dev. (*ms*)	0.000218	0.000305	0.000168	0.00267
	Confidence Int. (α=0.5)	4.91E-05	6.87E-05	3.77E-05	0.0006
PDA	Ave. (*ms*)	0.085880	0.092308	0.116889	0.145231
	Standard Dev. (*ms*)	0.001146	0.000985	0.005108	0.008006
	Confidence Int. (α=0.5)	0.000258	0.000221	0.001148	0.0018

From the results, we can see that the event notification time is much smaller than the architecture change time since all the event notifications occur in the same thread through messages. Further, the notification time is only related with the level of the hierarchical event tree, and not affected much by the tree structure. This is because an event source only notifies the subscribed upper layer event listeners, which creates a shortest message path from the simple event source located in the lowest level to the actuator and the path length is equal to the depth of the tree.

4.2.4 Message exchange time

MARCHES belongs to MOM and uses the publish/subscribe communication model. Message exchange time is defined as the time period between the time when the application data are input to the first marchlet and the time when the data are output from the last marchlet after they are processed by all marchlets of the active actuator. It stands for the communication efficiency of MARCHES. In this experiment, we only consider the scenario that all marchlets are located in the local agent. To support the distributed component communication, we can either migrate remote components to the local host or use RPC for remote communication, which belong to our future work. From the results in table 8, we observe that the time is just several milliseconds for message exchanges through 10000 components in a PC and 1000 components in a PDA, which is fairly small compared to the time for application data processing.

Table 8. Benchmarking MARCHES message exchange time.

	Comp #	1	10	100	1000	10000
PC	Ave. (*ms*)	0.001397	0.001583	0.004656	0.032029	1.423148
	Standard Dev. (*ms*)	0	0.00014	0.00014	0.00073	0.00433
	Confidence Int. (α=0.5)	0	3.14E-05	3.14E-05	0.000164	0.000973
PDA	Ave. (*ms*)	0.01535	0.018769	0.080308	1.466872	
	Standard Dev. (*ms*)	0.000447	0.001131	0.001359	0.00519	
	Confidence Int.l (α=0.5)	0.000101	0.000254	0.000305	0.001167	

4.3 Scalability (middleware performance in a distributed setting)

In MARCHES, a middleware agent maintains not only local proactive actuators, but also reactive actuators built for remote peer agents through the synchronization. Thus, the memory consumption is closely related with application scale. According to (1), the memory consumption R for a distributed vehicle applications is then modified as:

$$R = \sum_t (R_t) \ . \qquad (3)$$

where t is the index of peer middleware agents.

For example, if a vehicle application has 10 distributed programs deployed in vehicles and roadside infrastructure, each program contains 5 proactive actuators, 5 reactive actuators, each actuator has 10 marchlets, and each marchlet has 10 parameters, and if we use the default value setting in table 2, the memory consumption of each program is 5640 bytes × 10 (≈ 55 KB), which is still relatively small to most vehicle devices, like GPS.

5. Conclusion

In this paper, we have described a context-aware reflective middleware framework called MARCHES to support time-critical

adaptive vehicle systems. The overall objective of MARCHES is to improve the reconfiguration efficiency that has been realized by proposing a new adaptation structure of multiple component chains and a novel synchronization protocol using active messages to coordinate reconfigured behaviors asynchronously. Based on the architecture, MARCHES actuators perform reconfiguration actions by switching active and inactive component chains, which can reduce the local behavior change time compared to the traditional reconfiguration method of modifying the single chain architecture. Further, the asynchronous synchronization protocol dramatically reduces the reconfiguration time by eliminating the operation suspension and buffer clearance delays in the reconfiguration process in contrast to the traditional synchronous synchronization protocols in existing context-aware reflective middleware systems. Besides improving the efficiency, MARCHES offers some other benefits, including: (1) both component level and system level reflection for supporting the development of generic context-aware vehicle applications; (2) a binary tree based hierarchical event notification model for building efficient and comprehensive context-awareness sensors; and (3) an architecture-level description language that describes components, event sensors, actuators, and adaptation policies and a system-script development tool that facilitates the development of vehicle applications.

We have implemented and evaluated MARCHES in benchmark applications. The complete implementation of MARCHES and the applications allows us to test the memory footprint, time efficiency, robustness, and scalability of the middleware in vehicle and roadside environments, and gain insights into the adaptive and reflective middleware system design and behavior reconfigurations. The experiment results and analysis demonstrate that (i) the reconfiguration time in traditional adaptive and reflective middleware has been reduced by several magnitudes from seconds to hundreds of microseconds, (ii) the extra costs introduced by the multi-actuator architecture in MARCHES are extremely low comparing the hardware resources of vehicle devices, and (iii) the robustness and scalability are improved as well in MARCHES compared to traditional middleware.

The experiment results we have achieved so far are very positive, but there are some unexplored issues in MARCHES for future vehicle applications, which will be our future work.

- The component evaluation, identification, and migrations in the proposed component manager,
- The runtime reconfiguration of nonfunctional behaviors for complex vehicle systems, like the security strategies, communication styles, and routing protocols in mobile ad-hoc networks,
- The safe adaptation issue in behavior reconfiguration, especially when the behavior is related with history information, and
- The comprehensive deployment of MARCHES in real vehicle applications and experimental evaluations.

6. ACKNOWLEDGMENTS

The authors would like to acknowledge the support by the National Science Foundation (Award# 0438300).

7. REFERENCES

[1] Ray, R., Vehicle Infrastructure Integration Program Status, online document: http://www-nrd.nhtsa.dot.gov/pdf/nrd-01/NRDmtgs/2005Honda/Resendes_VII.pdf, retrieved on November 15, 2008

[2] Gene, M., CICAS Program Overview, TRB VII / CICAS Workshop, January 13, 2008

[3] Jiro, K., Advanced Cruise-Assist Highway System (AHS) Technology: System Design and Proving Test Facility Design, the 6th AHS Research Seminar, June 6, 2002

[4] Yasuyulu, A., Driving Safety Support System (DSSS) in the Aging Society, In: Proceedings of Intelligent Transportation Systems, Tokyo, Japan, 1999.

[5] CAR 2 CAR Communication Consortium, CAR 2 CAR Communication Consortium Manifesto, online document: http://www.car-2-car.org/fileadmin/downloads/C2C-CC_manifesto_v1.1.pdf, retrieved on November 15, 2008

[6] Matthias S., Tapani M., and Joachim I. etc., IP PReVENT Final Report, online document: http://www.prevent-ip.org/download/deliverables/IP_Level/PR-04000-IPD-080222-v15_PReVENT_Final_Report_Amendments%206%20May%202008.pdf, retrieved on November 15, 2008

[7] http://www.bts.gov/publications/national_transportation_statistics/html/table_01_11.html, retrieved on November 15, 2008

[8] http://www.lawcore.com/car-accident/statistics.html, retrieved on November 15, 2008

[9] Tarek A., Christopher D. G., Raj R., John A S., Distributed Real-time and Embedded Systems Research in the Context of GENI, NSF Workshop on Distributed Real-time and Embedded Systems, September 26, 2006

[10] Ron, B.N., CORBA: A Guide to Common Object Request Broker Architecture. McGraw-Hill, Inc. (1995)

[11] Troy, B.D., Java RMI: Remote Method Invocation. Foster City, Calif.: IDG Books Worldwide (1998)

[12] Thuan, L.T., Learning DCOM. Sebastopol, Calif.: O'Reilly, (1999)

[13] Alvin, T.C., Siu-Nam, C., MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing, In: IEEE Transaction on Software Engineering, 29(12) (2003)

[14] Licia, C., Wolfgang, E., Cecilia, M., Carisma: context-aware reflective middleware system for mobile applications, In: IEEE Trans. on Software Engineering, 29(10), pp. 929–945 (2003)

[15] Banavar, G., Ghandra, T., Strom, R.E., Sturman, D.C., A Case for Message Oriented Middleware, In: Proceedings of the 13th International Symposium on Distributed Computing, Bratislava, Slovak Republic (1999)

[16] Tai, S., Totok, A., Mikalsen, T., Rouvellou, I., Message Queuing Patterns for Middleware-Mediated Transactions. In: Proceedings of SEM 2002, Orlando, FL, Springer-Verlag (2003)

[17] Carzaniga, A., Rosenblum, D.S., Wolf, A.L., Design and Evaluation of a Wide-Area Event Notification Service, In: ACM Transaction on Computer Systems, 19(3), 332-383 (2001)

[18] Geihs, K., Middleware Challenges Ahead, In: IEEE Computer, 34(6) 24-31 (2001)

[19] Andersen, A., Blair, G.S., Stabell-Kulo, T., Myrvang, P.H., Rost, T.N., Reflective Middleware and Security: OOPP meets Obol, In: Proceedings of the Workshop on Reflective Middleware, Middleware 2003, Rio de Janeiro, Brazil; Springer-Verlag, Heidelberg, Germany (2003)

[20] Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., Saikoski, K. The Design and Implementation of Open ORB 2, In: IEEE Distributed System Online, 2(6) (2001)

[21] Object Management Group, CORBA Components OMG Document formal 02-06-65 (2002)

[22] Garbinoto, B., Guerraoui, R., Mazouni, K.R., Distributed Programming in GARF, In: Proceedings of the ECOOP Workshop on Object-Based Distributed Programming, Springer-Verlag, Kaiserslautern, Germany, pp. 225-239 (1993)

[23] McAffer, J., Meta-level Programming with CodA, In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Aarhus, Denmark, (1995)

[24] Cazzola, W., Ancona, M., mChaRM: a Reflective Middleware for Communication-Based Reflection. In: IEEE Distributed System On-Line, 3(2) (2002)

[25] Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhaes, L.C., Campbell, R.H., Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB. In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000), New York (2002)

[26] Thorsten, V.E., David, E.C., Seth, C.G. Klaus, E.S., Active messages: a mechanism for integrated communication and computation, In Proceedings of the 19th Annual International Symposium on Computer Architecture, pp. 256–266, Gold Coast, Australia (1992)

[27] Alan. M., David, C., Active Message applications programming interface and communication subsystem organization, In: Technical report, Dept. of Computer Science, UC Berkeley, Berkeley, CA (1996)

[28] Alan. M., David, C., Design challenges of virtual networks: Fast, general-purpose communication, In: Proceedings of the 1999 ACM Sigplan Symposium on Principles and Practise of Parallel Programming (PPoPP‘99), vol. 34.8, pp. 119–130, A.Y. (1999).

[29] Eduardo, S., Germano, G., Glauco, V., Mardoqueu, V., Nelson, R., Carlos, F., A Message-Oriented Middleware for Sensor Networks, In: Proceedings of the 2nd Workshop on Middleware For Pervasive and Ad-Hoc Computing, Toronto, Ontario, Canada, (2004).

[30] Qusay, H., Middleware for Communications, 1st ed., John Wiley & Sons Ltd., Chichester, England (2004)

[31] Qiang, W., Liang, C., AwareWare: an Adaptation Middleware for Heterogeneous Environments, In: Proceedings of 2004 IEEE International Conference on Communications (ICC 2004), Vol. 3, pp. 1406-1410, Paris, France (2004)

[32] Justin, M. P., Hubert, P., Umar, S., Grace, C., Chris, T., Steve, W., Structured Decomposition of Adaptive Applications, In: Proceedings of the 6th IEEE International Conference on Pervasive Computing and Communication (PerCom) (2008)

[33] Stefanos, Z., Cecilia, M., The SATIN Component System—A Metamodel for Enginnering Adaptable Mobile System, In: IEEE Trans. on Software Engineering. 32 (11) (2006)

[34] Radu, L., Atul, P., DACIA: A Mobile Component Framework for Building Adaptive Distributed Applications, In: Technical Report CSE-TR-416-99, University of Michigan, EECS (1999)

[35] Object Management Group, CORBA Component Model Joint Revised Submission, OMG Document orbos/99-07-01.

[36] Uwe B., Aurelie B., Florentin P., and Etienne S., Distributed Real-Time Computing for Microcontrollers - The OSA+ Approach. In Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, page 169. IEEE Computer Society, 2002.

[37] Schmidt D., Huston S., C++ Network Programming: Resolving Complexity with ACE and Patterns, Addison-Wesley, Reading, MA, 2001.

[38] Raymond K., Douglas C. S., Carlos O. Towards highly configurable real-time object request brokers. In Proceedings of ISORC’02, pages 437–447, 2002.

[39] Douglas C. S., David L. L., and Sumedh M., The design of the TAO real-time object request broker. Computer Communications, 21(4):294–324, Apr. 1998.

[40] Venkita, S., Guoliang, X., Christopher D., and Cytron, R., The Design and Performance of Special Purpose Middleware: A Sensor Networks Case Study, 2003. Technical report of the University of Washington Saint Louis 2003, # 6.

[41] OSEK Comitee. OSEK/VDX Home Page, 2004. http://www.osek-vdx.org/

[42] Garbinato, B., Guerraoui, R., and Mazouni, K.R., Distributed Programming in GARF, Proceeding of the ECOOP Workshop on Object-Based Distributed Programming, Springer-Verlag, Kaiserslautern, Germany, pp. 225-239, 1995

[43] McAffer, J., Meta-level Programming with CodA. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Aarbus, Denmark, 1993

[44] Gordon S. B., Geoff C., Anders A., The Design and Implementation of Open ORB 2. IEEE Distributed Systems Online 2(6): ,2001

[45] Fabio K., Manuel R., and Ping L. etc., Monitoring, Security, and Dynamic Configuration with the DynamicTAO

Reflective ORB, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.

[46] Zinky, J. A., Bakken, D. E., and Schantz, R., Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems, 3(1), 1-20. 1997