A Lightweight Component-Based Reconfigurable Middleware Architecture and State Ontology for Fault Tolerant Embedded Systems

Jagun Kwon and Stephen Hailes

Department of Computer Science
University College London, Gower Street, London, WC1E 6BT, UK
{j.kwon,s.hailes}@cs.ucl.ac.uk

Abstract. In this paper, we introduce a component-based software architecture that facilitates reconfigurability and state migration in a semantically correct manner for fault tolerant systems. The main focus and contribution of the paper is on the ontology framework that is based on object orientation techniques for coherent reconfiguration of software components in the events of faults at runtime, preserving state consistency and also facilitating state ontology evolution. Our proposed approach is realised in C++, which is integrated in the lightweight middleware MIREA.

MIREA is a reconfigurable component-based middleware targeted at embedded systems that may not have abundant resources to utilise, such as sensor systems or embedded control applications. The middleware supports software componentisation, redundancy and diversity with different software designs in order to ensure the independence of common operational/development errors. Moreover, any unforeseen errors can be dealt with by dynamically reconfiguring software components and restoring states.

Keywords: Component-based middleware, Reconfigurability, Embedded systems, State, Ontology, Fault tolerance.

1 Introduction

Today, software accounts for up to 40% of the total production cost in many embedded systems due to the complex, inflexible and proprietary nature [2]. This means that the cost of poor or repeat engineering in this area is extremely high and that flexibility, reusability and reconfigurability are key factors in staying competitive in the market.

Complexity can be managed most effectively if the underlying software systems support structured, standardised, high-level abstraction layers that encapsulate unnecessary details behind well-defined interfaces; this has the effect of reducing training effort, development time and cost. Moreover, since the cost of software maintenance is often as high as the cost of initial development, the ease with which it is possible to flexibly deal with faults and reconfigure software components in operational systems is also critical. In this sense, the use of a middleware is an attractive solution.

In this paper, we describe a component-based software architecture that facilitates reconfigurability and state migration in a semantically correct manner for fault tolerant embedded systems. The main focus and contribution is on the ontology framework that is based on object orientation techniques for coherent reconfiguration of software components in the events of faults at runtime, preserving state consistency and also facilitating state ontology evolution. Our proposed approach is realised in C++, which is integrated in the component-based middleware MIREA [19].

The middleware has been developed to address some of the important requirements in the development of modern embedded systems, such as support for predictability, reconfigurability, fault-tolerance, portability for heterogeneous platforms, object oriented programming, and component-based composition, to name a few.

MIREA facilitates software reuse and runtime reconfigurability to reduce development and maintenance cost and to fight unforeseen faults. States can be migrated in a coherent manner as long as correct transformation logics are specified by following the proposed model in the paper.

In the remainder of this paper, we first describe the middleware's component model and kernel's key features such as component reconfiguration/rebinding. Section 3 presents our proposed model of state ontology for consistent state migration and state ontology evolution based on object orientation techniques. The next section briefly reviews related work before drawing a few conclusions.

2 The MIREA Middleware

Middleware is commonly defined as a software layer that lies between the operating system and application software, often in distributed systems. Its importance has been growing not only in enterprise systems, but also in embedded systems, where reuse of legacy code and standard development environments are becoming increasingly important. According to [3], there are four functions of middleware:

- *Hiding distribution, i.e., the fact that an application is usually made up of many interconnected parts running in distributed locations should not be apparent;*
- *Hiding the heterogeneity of various hardware components, operating systems and communication protocols;*
- *Providing uniform, standard, high-level interfaces to the application developers and integrators, so that applications can be easily composed, reused, ported, and made to interoperate;*
- *Supplying a set of common services to perform various general purpose functions, in order to avoid duplicating efforts and to facilitate collaboration between applications.*

In other words, the use of middleware makes software development easier, more reliable and cost-effective by providing common programming abstractions, hiding low-level software and hardware details, and facilitating reuse of established, proven components. In the context of embedded systems, the use of middleware is also well justified due to the cost of development and the bespoke nature of the systems. However, since most embedded systems are resource-conscious or constrained, light-weight systems are much to be preferred. Along with reconfigurability, scalability and real-time quality-of-service features, this has been one of the key guiding principles in the development of the MIREA middleware.

2.1 Component Model

MIREA assumes a generic contract-based component model, in which a component can specify functionality that it requires and/or provides using a well-defined interface. The model consists of *ComponentTypes*, *Components*, *Interfaces*, *Receptacles*, *States* and *Connectors*. A *Component* is a runtime instance of a *ComponentType*. A *ComponentType* can export one or more *Interfaces*, through which a given component provides a set of functionalities to other components (i.e., in the form of a set of C/C++ functions in the middleware). A *Component* can have any number of *Receptacles*, through which a set of required functionalities are specified. Figure 1 below illustrates the elements of the component model.

A component can also have an associated component-wide state that is only accessible from within the containing component; a state can consist of a set of any data types and/or primitive variables. *Connectors* are a specialised form of component that performs intermediary actions if required, for instance, in order to monitor, log or encrypt data for security reasons. Note that a connector is simply a component and, as such, can be realised as a composite component, with component state and connectors of its own if this makes sense.

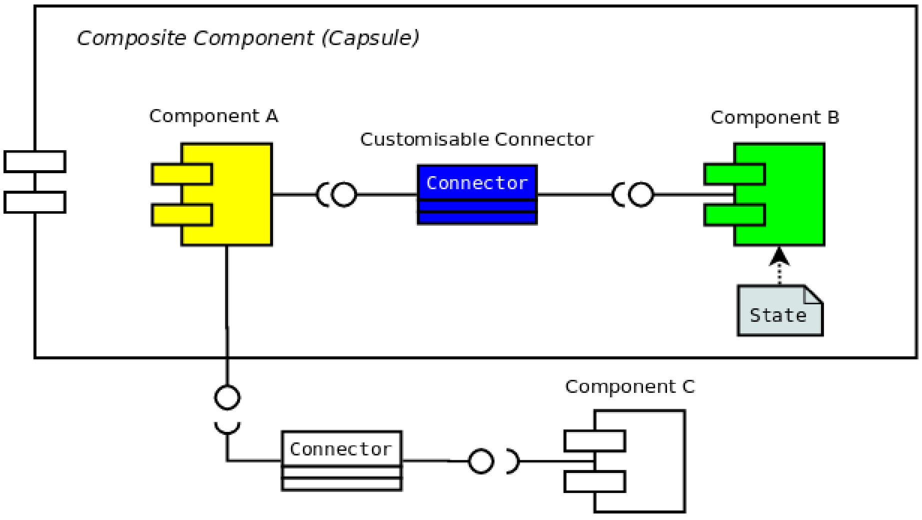


Fig. 1. Exemplified Component Model Elements

2.2 Middleware Kernel

The middleware kernel provides a framework for realising the component model in practice, so that component types can be defined with implementation details and a set of receptacles and interfaces in the framework, and they can be instantiated using the middleware's services at run-time. Underneath the exposed middleware API, there can be certain cross-platform components that make use of operating system services or hardware abstraction layers. User space interrupt handlers and/or hardware drivers may be situated at this layer or embedded within the operating system. On top of the middleware API lies either generic or specific services for embedded applications.

2.3 Core Services

The vast majority of embedded software is written in C and C++. By supporting C++, a large body of existing software libraries, such as NIST's RCS NML [1] can be reused, integrated seamlessly with other components running on this middleware, and used to build new applications cost-effectively while taking advantage of the object oriented design and programming methodology. In order to support such flexibility and scalability without incurring excessive runtime overheads, the middleware provides the following categories of core services:

- Loading and unloading of *ComponentTypes*
- Instantiation and destruction of *Components*
- Registration and acquisition of Interfaces and *Receptacles*
- Connection between *Interfaces* and *Receptacles*
- Registration and acquisition of *Components'* States
- Destruction of *Connectors*
- Check-pointing of *States*
- Support for *State* evolution

All of the services above are runtime activities whereas the process of defining a new *ComponentType*, *Interface*, *Receptacle*, *Connector* and *State* are static in nature - defined at an application design stage where the user may consider communication overheads between components and complexity of the application composition. Connections between components can be reset and *reconfigured* by first destroying an existing connector instance and then reconnecting them to a new type of interface and receptacle. After this, any invocations made on the given *Receptacle* will be redirected to the newly connected *Interface* instance, hence a new/different *Component* instance.

2.4 Inter-Component calls, Interfaces and Receptacles

The middleware allows pure virtual C++ classes to be used as *Interfaces* and *Receptacles*. In C++, pure virtual classes only define the prototypes and signatures of functions that must be implemented in an inheriting class if they are to be instantiated as objects. By doing so, we enhance type-safety of applications by ensuring that pairing between interfaces and receptacles is type-checkable in C++. Figure 2 illustrates this point, where two components A and B agree on the use of the common interface for their *Receptacle* and *Interface*. Component A expects to call operations *op1* and *op2* while component B implements the relevant operations and provides an interface to that by inheriting the agreed class and instantiating an *Interface* object. After connection, whenever A calls functions in the *Receptacle*, those of the matching *Interface* object will eventually be invoked.

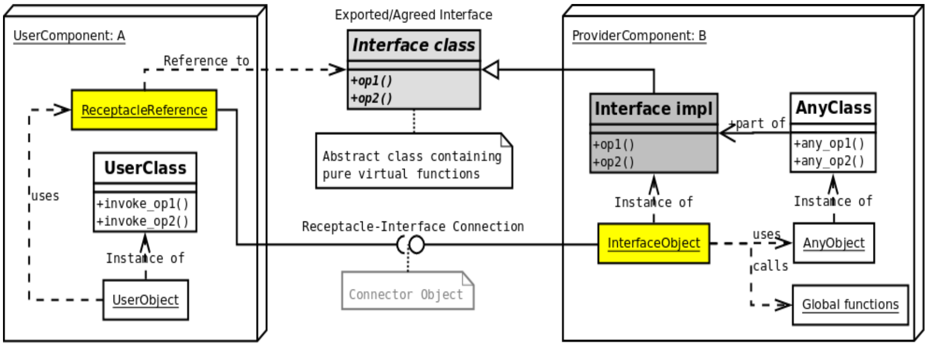


Fig. 2. Component Connection Model

Type-checking of an arbitrary component is achieved by making use of Xrtti [15], an automated reflection mechanism for C++, which ensures that users are not required manually to provide introspective class information.

2.5 Component Implementation and Migration of Legacy Code

There is bound to be reusable legacy code in many software engineering projects, and migration efforts for such code must be kept to the minimum at any time. Creating a MIREA component is a straightforward process where one can either reuse existing code or create a new C/C++ project implementing all the required interfaces and receptacles with the middleware's conventions in mind, and compile and link the code into a dynamic shared library (for instance, *.so* files in case of Linux or Unix). The middleware requires every component to provide a pair of *construct*() and *destruct*() functions, and register the interfaces and receptacles they operate on in the *construct*() function. The *destruct*() function is used to de-allocate reserved memory for states or any other objects within the component. So long as a piece of code follows these undemanding rules, it can be seen and used as a MIREA component. Because of the simplicity, the middleware has a negligible impact on the performance of the whole system. Refer to [19] for the overheads and performance figures.

Depending on the architecture of the deployed system, components may reside locally or be downloaded to embedded nodes through network, loaded and utilised using the middleware's services. Managing the location of a component is up to the system's designer and should be strategically handled, for instance, to reduce network traffic or lower security risks.

3 State Ontology for Check-Pointing and Reconfiguration

3.1 Problem Description

The ability to reconfigure software components dynamically is beneficial in that it provides flexibility in design and implementation while reducing development cost by means of reusing proven artefacts. One can also delay design decisions and opt for alternatives in case of faults at runtime or as a result of changes in requirements.

State Migration Problem

However, as software components are dynamically reconfigured or replaced in an embedded system, there are situations where ***previous states must be preserved in a semantically correct manner and transferred to the new replacing components*** in order to ensure correct, smooth transition of operation. There is no standard way of doing this in widely used languages like C/C++ and any middleware for embedded systems, i.e., there is no schema- or ontology-based approaches to migration of states.

It is also possible for knowledge structures or ontologies (we use the term ontology rather informally and it can be interchanged with *schema* here) to evolve during the life of the system, and hence, there is a need for

- A means for expressing/representing ontology for knowledge base and *instances* for the purpose of storing and querying about states,
- A means to extend ontology and reason about changes between different versions, and
- A lightweight programming interface for accessing and reasoning about such knowledge base schemas and instances.

Illustration

Imagine that there is a software component called *CPathPlanner2009* that deals with path planning for a laser-guided vehicle (LGV) in a factory. *CPathPlanner2009* implements algorithm *a*, which is later found to be inefficient and to be replaced with a newer one. Hence, a newer component *CPathPlanner2010* that implements algorithm *b* is about to be deployed while the LGV is in operation or recharging its batteries. *CPathPlanner2009* currently holds information regarding the destination and source of pallets, locations of several via-points and charging stations, number of other LGVs operating at the same time, and factory floor plan. Along with this static information, it also stores various intermediate values and results of calculation regarding where it is heading towards, current speed of the vehicle, various sensor values (e.g., collision sensor values), distance to the nearest via-point and so on. Such information may need to be transferred and reused in a semantically correct manner in order to prevent abrupt operation or restart of the LGV's controller system.

However, some of the state information may be unnecessary or have different meanings in different components; for instance, algorithm *a* requires the number of other LGVs while algorithm *b* does not because it relies more on the sensory input values than pre-defined world-model parameters.

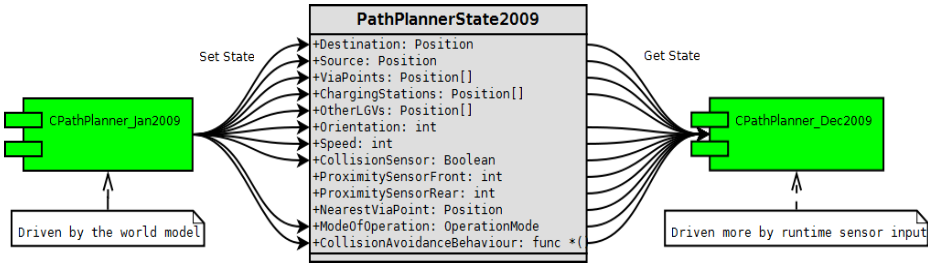

PathPlannerState2009
+Destination: Position
+Source: Position
+ViaPoints: Position[]
+ChargingStations: Position[]
+OtherLGVs: Position[]
+Orientation: int
+Speed: int
+CollisionSensor: Boolean
+ProximitySensorFront: int
+ProximitySensorRear: int
+NearestViaPoint: Position
+ModeOfOperation: OperationMode
+CollisionAvoidanceBehaviour: func *(
Set State
Get State
CPathPlanner_Jan2009
CPathPlanner_Dec2009
Driven by the world model
Driven more by runtime sensor input

Fig. 3. State Migration Example

State Ontology Evolution

The structure of the state may also evolve during the course of system development and the lifetime of the components. In other words, the state information that is stored, checkpointed, or required by a new component may differ from the one currently used, meaning that it is possible to make changes to the existing state information structure.

Hence, it is important to be able to track changes in state structures in order to ensure information compatibility; for instance, an existing component may still use an old state structure to store and transfer state information to a new component. This new component must be able to interpret the previous state structure automatically, and in order to do that, it has to reason about the changes made between the different versions of the state structure.

This is also useful in *check-pointing* for fault tolerant systems that store states at a regular time interval or at sporadic requests. States must be stored in a format that can be interpreted by other components that may attempt to recover from faults or errors at a later stage using one of the previously stored states.

3.2 Proposed Approach: State Ontology

First, let us consider the fundamental motivations for developing ontology, i.e., to share common understanding of the structure of information among people or software agents [17]. This also implies reuse and analysis of domain knowledge. In practice, developing an ontology involves [18]

- Class definition in the ontology,
- Arrangement of the classes in a taxonomic hierarchy (i.e., subclass–superclass),
- Definition of slots or attributes and describing allowed values for these slots,
- Instantiation by filling in the values for slots.

It is our observation that this can be captured in object-oriented programming languages although operational knowledge is more focused.

Proposed Approach

We believe that object orientation techniques realised in the form of C++ classes can be used as a means of expressing, maintaining and reasoning about ontology, especially in the context of software development for fault tolerant embedded systems that must ensure state consistency. Take *check-pointing* for example; states must be stored constantly in the right form, preserving consistency. That way, if faults develop in one of the current components, a new component, which is unknown at the time of dispatch, can restore the states and operate smoothly after the faults are isolated and subject components are reconfigured using the middleware's services. In such a process, there can be more than one type of check-pointed states that a component or software agent has to deal with. Thus, states should be coherently organised along with transformation rules or logics.

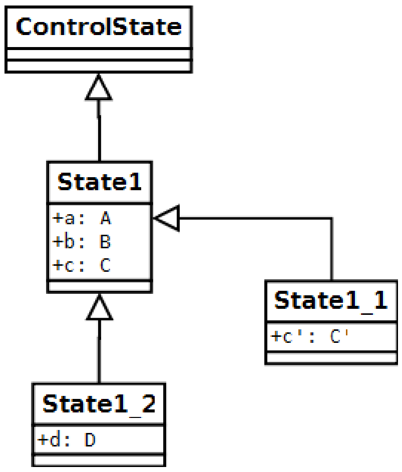


Fig. 4. State ontology evolution realised in the form of class hierarchy

The reasons for using C++ instead of other declarative languages are as follows.

- It is easy to adopt in programming terms. One can even embedded operational knowledge by means of C++ objects and functions since certain type of knowledge can only be expressed in programming terms.
- Less overhead involved in reasoning and accessing knowledge base instances (i.e., objects), thus suitable for resource and performance-sensitive applications (especially considering the middleware itself is written in C++).
- Schema/ontology evolution is possible based on object orientation techniques by means of inheritance and encapsulation.

Possible types of ontology evolution in C++ are

- *Addition of new fields and functions*: Inheritance with new members
- *Changes of fields and functions:* Overriding of relevant fields and functions
- *Removal or change of semantics in functions*: Nullifying or redirecting with a wrapper that overrides a function

However, there are some downsides of using a procedural language, i.e.,

- Expressive power of the language may be limited.
- One may have to depend on a reflection mechanism in order to reason about ontology/schema at runtime, which could cause extra overheads at runtime or be complex to learn.
- It is difficult to learn or understand for those who are not familiar with C++ or object-orientation concepts.

But, since we are dealing with embedded software engineers who are likely to be familiar with programming in C or C++, the last point is not a real obstacle. C++ has been widely used by not just computer scientists, but also by a wide range of scientific and engineering communities, and the language is standardised. In addition, it is not necessary to learn the whole language, but only the class-related subset, which is reasonable and comparable to learning a different ontology language, such as OWL and Protégé. Any tools that can generate code from UML class diagrams could also be employed in constructing ontology in C++.

Expressive power of the language may seem limited at first compared to other purpose-built languages. But, for the purpose of storing readily usable information specific to the embedded systems domain, we believe that C++'s expressive power is adequate. As a bonus, behavioural or operational knowledge in the form of functions can be captured as well.

Moreover, reflection mechanisms for C++ are available in a few different flavours, for instance, ones that make use of templates and macros to document extra information about classes within source code, compiler and debugger-based approaches and so on. We find that Xrtti [15] is one of the most convenient means of accessing meta-class information at runtime. It provides libraries and APIs that facilitate users to access meta-class information at runtime. Its front-end takes original C++ header files and generates Xrtti-compliant headers that need to be included in the code that invokes Xrtti's APIs to access metaclass information.

Usage of the Proposed Model

State ontology is of use in

- State migration in fault-tolerant systems (e.g., when recovering from faults in a reconfigurable control system, if a component fails, its state could be transferred to a new, different component) or
- Check-pointing of component states and restoring them if necessary

The conceptualisation or specification of an ontology may change over time as we progress with or become better aware of the problem domain. However, to preserve the ontology's consistency one must keep track of changes in the ontology as it is updated. Especially, when multiple parties (such as, multiple threads or even distributed agents) are communicating with each other assuming different knowledge representations and versions, they will eventually conflict with each other unless they are consistently mapped and changes and differences are resolved. This can happen when a component with a state ontology is replaced, due to software faults, with another component that assumes a different version of the state ontology. Interoperability can be supported by keeping track of changes and converting states into different versions when required.

A state ontology in C++ can evolve by means of inheriting a parent class and overriding attributes and functions, or even by *nullifying* by overriding a function with an empty function or redirecting calls to a correct function to remove or keep consistency. In other words, according to the inheritance hierarchy that can be seen as a version history, changes between parent and child classes can be tracked and reasoned about.

To illustrate this point, imagine that component *CPathPlanner2009* has developed a fault and the most recent state has been checkpointed in state *s* that is an instance of *PathPlannerState2009* (see Figure 5 below for an illustration). An alternative component *CPathPlanner2010* exists, but it has been designed to work with a more evolved ontology *PathPlannerState2010*, which contains additional members and certain changes in the semantics that were not foreseen and are now part of new state instances. Each attribute has an associated *getter* and *setter* function, some of which the updated state ontology overrides and nullifies with additional transformation logics for the different version to convert to the immediate parent state and *vice versa*. In order to reduce downtime, the old component is replaced dynamically with the new one, taking the most recent stable checkpointed state. Since the state from the old component is the parent of the new state ontology class, they are compatible to each other although certain changes are present on the new one. The mechanisms to convert an old state into a new one are embedded in the new, overriding, or nullifying functions. For instance, if a state instance of *PathPlannerState2009* is adopted in the place of *PathPlannerState2010*, and function *getDestination* is updated by overriding it, the new version of *getDestination* may contain a program logic that redirects calls to its parent (or super) class when it needs to obtain or associate state values with its previous version. In other words, if the migrated state instance is of type *PathPlannerState2009* (i.e., the old one), calls to *getDestination* can be redirected conditionally to the original one, and converted into a new state value in a semantically correct manner given a correct transformation logic within the new function.

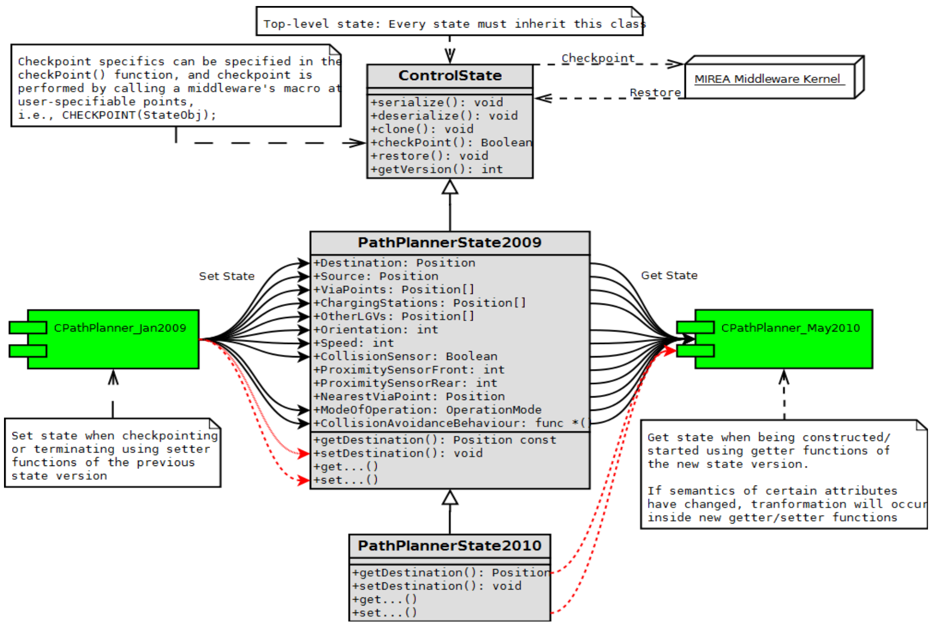
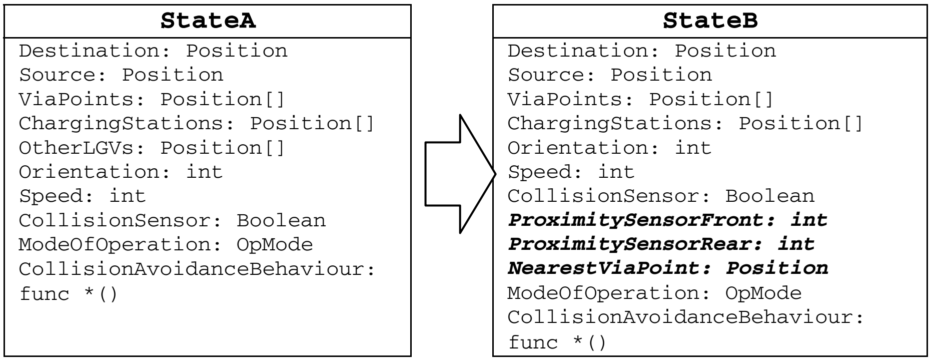

Top-level state: Every state must inherit this class
Checkpoint specifics can be specified in the checkPoint() function, and checkpoint is performed by calling a middleware's macro at user-specifiable points, i.e., CHECKPOINT(StateObj);
ControlState
+serialize(): void
+deserialize(): void
+clone(): void
+checkPoint(): Boolean
+restore(): void
+getVersion(): int
Checkpoint
Restore
MIREA Middleware Kernel
PathPlannerState2009
+Destination: Position
+Source: Position
+ViaPoints: Position[]
+ChargingStations: Position[]
+OtherLGVs: Position[]
+Orientation: int
+Speed: int
+CollisionSensor: Boolean
+ProximitySensorFront: int
+ProximitySensorRear: int
+NearestViaPoint: Position
+ModeOfOperation: OperationMode
+CollisionAvoidanceBehaviour: func *(
+getDestination(): Position const
+setDestination(): void
+get...()
+set...()
Set State
Get State
CPathPlanner_Jan2009
CPathPlanner_May2010
Set state when checkpointing or terminating using setter functions of the previous state version
Get state when being constructed/ started using getter functions of the new state version.
If semantics of certain attributes have changed, tranformation will occur inside new getter/setter functions
PathPlannerState2010
+getDestination(): Position
+setDestination(): void
+get...()
+set...()

Fig. 5. Illustration of the proposed state ontology framework

Implementation for State Transformation

The idea above can be realised using a *copy constructor* in C++ as briefly sketched below and a piece of skeleton code is shown. Assume that there are state ontology classes *StateA* and *StateB* for the sake of simplicity. When an object of *StateA* is passed between two components (where one is replaced by the other) and transformed into more specialised *StateB* for use in the new component, more information may need to be deduced along with the original contents of *StateA*. For instance, components utilising *StateB* may be driven more by runtime sensor input than relying on a static world model, and hence there are additional information fields for storing sensor values in *StateB*. However, a large set of data fields are inherited when converting *StateA* into *StateB*.



In other words, when *StateA* is converted into *StateB*, the following copy constructor can determine automatically what information is to be kept while what and how other information needs to be deduced depend on the version of the source state object that is passed in as a parameter to the copy constructor. See the case for **`(i.getVersion() < MAJOR_VERSION),`** where some of the values are assigned normally while other values are deduced by extra calculations (for instance, the representation of locations were based on a static world model, but now incorporates GPS input).

However, when such a conversion is performed backward, i.e., from a more specialised child state into a parent one, it is always the responsibility of the newer child class to convert its type into a previously defined parent type correctly – such information is encoded in a user provided function *convert*().

```
class StateB {
  const int MAJOR_VERSION=2;// class-wide version of this state type
  const int MINOR_VERSION=0;
  …
  // Copy constructor
  // Check the version of each class when assigning one to
  // the other, and determine how they should be converted.
  StateB(const StateB& i) {
  // First check if given state i belongs to the same state hierarchy
    …
    if (i.getMajorVersion() == MAJOR_VERSION) {
      // No type conversion is required, just plain copy of states.
      setDestination(i.getDestination());
      setSource(i.getSource());
      // Same for other attributes …
    }
```

```
      else if (i.getVersion() < MAJOR_VERSION) {
        // If the version of i is lower (i.e., one of the parents),
        // convert i into this object's type.
        if (I.getVersion() == MAJOR_VERSION-1) {
          //Assume the locations are encoded differently in StateB,
          setDestination(i.getDestination() * getCurrentGPS() * x);
          setSource (i.getSource() * getCurrentGPSPos() * y);
          setViaPoints (i.getViaPoints());
          // may need further calculations to deduce values…
        }
        else if (…) {}
        // This conversion logic could vary depending on the versions
      }
      else if (i.getVersion() > MAJOR_VERSION) {
        // If the version of i is higher (i.e., one of the children),
        // delegate the type conversion task to i (thus, i into this).
        // It is always the newer(child) class that knows how to
        // convert its(child) type into an previous(parent) type.

        tmp = i.convert(MAJOR_VERSION);
        // i is converted into a different version
        // (i.e., to this version)
        // if i is of type StateC, StateC's convert() will be called
        // and copy itself into a requested VERSION of the type,
        // by calling its super classes recursively if necessary
      }

      // Object i may be freed up after state migration
  }

  ControlState convert(int ver) {
    // If a backward state conversion is requested (into a lower ver.)
    if (ver < MAJOR_VERSION) {
      return static_cast<StateA>(this).convert(ver);
    }
    else if (ver == MAJOR_VERSION) {
      // Now clone/deep-copy and return the object's reference
      //…
      return *lowerCloned;
    }
  }
  }
```

Middleware's Support for State Ontology

The middleware defines a special class named *ControlState*. This is the top-level framework that every state ontology class must inherit. By overriding the member functions, the user can specify how a specific state instance can be serialised, deserialised, cloned, check-pointed, and restored. The middleware also provides a special macro called *CHECKPOINT*, which can be used to specify where and when the user wants to checkpoint states within the application code. When the middleware encounters a call to *CHECKPOINT*, it will call the subject state's *checkPoint*() function to store the state, which is expected to be provided by the user or inherited from a parent class.

4 Related Work

There are a large number of middleware systems that have been developed for different purposes. However, for resource-conscious embedded systems, one often settles with less dynamic or less flexible options because of predictability and safety concerns. In most cases, it is also logical to rule out heavy middleware systems that can cause longer, unpredictable delays and are difficult to reason about. Generally speaking, CORBA-based middlewares tend to be large in memory footprint and unfavourable for resource-constrained embedded applications.

Compared to the RUNES [7], MIREA[19] supports C++ components, type-checking and QoS-related reasoning. OROCOS [10] is specifically designed for robotics applications. It comes with kinematics and dynamics libraries and real-time tool kits. Its component model is comprehensive and provides a rich set of features including scripting. For lightweight embedded systems like wireless sensor systems, MIREA has a smaller footprint and better flexibility.

In [13] and [14] build-level component models are discussed. Although targeted at consumer electronics software, they support variability/reconfigurability at build-time, and are lightweight in terms of development effort. However, they do not provide a runtime middleware kernel, or a state migration framework.

5 Conclusions and Future Work

In this paper, we introduce a component-based middleware specifically designed for resource-constrained embedded systems, and its proposed state migration approach for fault-tolerant systems. MIREA has a small memory footprint (34KB) and low runtime overheads [19], yet is component-based. It facilitates software reuse and runtime reconfigurability to reduce development and maintenance cost and to fight unforeseen faults. States can be migrated in a coherent manner as long as correct transformation logics are specified by following the proposed model in the paper.
Currently, we are in the process of applying the proposed model in a real scenario that involves LGVs (laser guided vehicles) in a factory. As a future work, we will evaluate the effectiveness of the approach and the middleware, and report on the findings.

Acknowledgments. This work has been supported by the Seventh Framework ICT Programme of the European Union under contract number ICT-2007-224428 (CHAT).

References

[1] Gazi, et al.: The RCS Handbook, Tools for Real-Time Control Systems Software Development. John Wiley & Sons, Inc., Chichester (2001)

[2] NGMS IMS (Next Generation Manufacturing Systems-Intelligent Manufacturing System) Research Reports, "Scalable flexible manufacturing", Advanced Manufacturing (March 2000), `http://www.advancedmanufacturing.com/March00/research.htm`

[3] Krakowiak, S.: What is Middleware, ObjectWeb, A more complete version available as ”Middleware Architecture with Patterns and Frameworks” (2003), `http://middleware.objectweb.org/`, `http://sardes.inrialpes.fr/~krakowia/MW-Book/Chapters/Intro/intro.html`
[4] Xenomai Official Website, `http://www.xenomai.org/`
[5] Morton, Y.T., Troy, D.A., Pizza, G.A.: An Approach to Develop Component-Based Control Software For Flexible Manufacturing Systems. In: Proc. of the American Control Conference, Anchorage, AK, May 8-10 (2002)
[6] Stroustrup, B.: The C++ Programming Language (1997)
[7] RUNES, `http://www.ist-runes.org`
[8] Costa, P., Coulson, G., Mascolo, C., Picco, G.P., Zachariadis, S.: The RUNES Middleware: A Reconfigurable Component-based Approach to Network Embedded Systems. In: Proc. of 16th International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC 2005). IEEE Press, Los Alamitos (2005)
[9] NIST, 4D/RCS: A Reference Model Architecture for Unmanned Vehicle Systems, Version 2.0, `http://www.isd.mel.nist.gov/projects/rcs/ref_model/coverPage3.htm`
[10] OROCOS, `http://www.orocos.org/`
[11] Szyperski, C.: Component Software – Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Reading (2002)
[12] Szyperski, C.: Component Technology: what, where, and how? In: Proc. of the 25th Intl. Conference on Software Engineering (2003)
[13] Ommering, R., Linden, F., Krammer, J., Magee, J.: The Koala Component Model for Consumer Electronics Software. IEEE Computer 33(3) (March 2000)
[14] Park, C., Hong, S., Son, K., Kwon, J.: A Component Model supporting Decomposition and Composition of Consumer Electronics Software Product Lines. In: Proc. of 11th IEEE Intl. Software Product Line Conference (2007)
[15] Xrtti: Extended Runtime Type Information for C++, documentation and download, `http://www.ischo.com/xrtti/`
[16] Gat, E.: On the Role of Stored Internal State in the Control of Autonomous Mobile Robots. AI Magazine 14(1) (1993)
[17] Gruber, T.R.: A Translation Approach to Portable Ontology Specification. Knowledge Acquisition 5, 199–220 (1993)
[18] Stanford University, Ontology Development 101: A Guide to Creating Your First Ontology (Protégé Documentation), `http://protege.stanford.edu/publications/ontology_development/ontology101.html`
[19] Kwon, J., Hailes, S.: MIREA: Component-based Middleware for Reconfigurable, Embedded Control Applications. In: The Proc. of the IEEE International Symposium on Intelligent Control, ISIC (2010)