X-MAN: An MDE Tool for Component-based System Development

Kung-Kiu Lau and Cuong M. Tran
School of Computer Science, The University of Manchester
Manchester M13 9PL, United Kingdom
Email: kung-kiu, ctran@cs.manchester.ac.uk

***Abstract*—The X-MAN tool has been developed in the European project CESAR, for component-based system development. CESAR is a large industrial project that aims to develop a component-based technology that can be used for developing embedded systems in multiple domains, including automotives and avionics. X-MAN has been successfully evaluated by CESAR's external reviewers and internally by Airbus Operations Limited for the avionics domain. X-MAN is designed and implemented using MDE; in this paper we describe its design and implementation.**

I. Introduction

Component-based system development is one of the main technical objectives of the CESAR project (see footnote). Indeed CESAR's aim is to develop a component-based technology to support system development for embedded systems in multiple domains: automotives, avionics, railways, aerospace and automation. As a partner in CESAR, we (the University of Manchester) have developed a tool for component-based system development. This tool is based on the X-MAN component model [8], and supports the complete life cycle for component-based system development [10]. The X-MAN tool has been successfully evaluated by external reviewers of the CESAR project as well as by Airbus Operations Limited for the avionics domain within CESAR.

X-MAN is an MDE tool, implemented in GME [14]. In this paper, we discuss the design and implementation of X-MAN.

II. Component-based System Development

Component-based development aims to (i) build components and deposit them in a repository; and (ii) use or reuse these pre-existing components to build many different systems by assembling the components using composition mechanisms that have also been pre-defined [9].

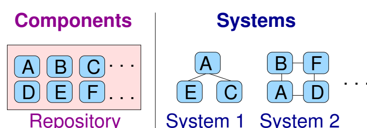

Fig. 1. Component-based system development.

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement no. 100016 and from the Technology Strategy Board, UK. It has been carried out on the CESAR project (http://www.cesarproject.eu).

This is illustrated by Fig. 1. For a given problem domain, components are identified and developed, using domain knowledge. Similarly, composition mechanisms for the components are defined and fixed for the domain.

In adddition, V&V (Verification and Validation) should be carried out on components in the repository, prior to deploying these components in different systems. That is, component V&V should be performed separately from, and prior to system V&V.

A. The W Model

The life cycle for component-based system development is thus different from that for traditional approaches. The standard model for traditional approaches is the V model. The V model is applied to only one system; it defines a top-down design process that starts from the system specification and arrives at a system architecture by successively decomposing the specification. Components in the system architecture are thus identified and defined *top-down*; these components are specific to this system, and are therefore not repository components. Thus the V model defines only a system life cycle, not a component life cycle. In contrast, in component-based development, components are identified and defined *bottom-up*, from domain requirements. These components are domain-specific, but not specific to any one system, and are therefore repository components developed in a component life cycle, separately from, and prior to, the system life cycle.

Therefore a complete life cycle for component-based development should consist of two life cycles: a *component* life cycle and a *system* life cycle. Accordingly, in [10], we proposed the W model for component-based development. As depicted in Fig. 2, the W model essentially consists of two V models: one for the component life cycle and one for the system life cycle. The component life cycle starts from domain knowledge and ends with a (domain-specific) component repository [1]. The system life cycle, starts from system requirements, takes components from the component repository developed in the component life cycle, and ends with acceptance testing. V&V for components happens in the component life cycle, separately from, and prior to, system V&V in the system life cycle. Indeed, due to component V&V, system V&V should be much easier, since it can make use of

[1]The component life cycle may be repeated for all components in a domain.

978-0-7695-4790-9/12 $26.00 © 2012 IEEE
DOI 10.1109/SEAA.2012.32

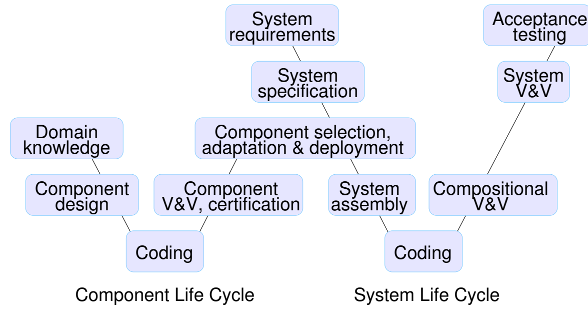

Fig. 2. The W Model.

compositional V&V, which results from the composition of components that have been V&V'ed previously.

To develop repository components in the component life cycle, and to deploy them into systems in the system life cycle, we need a component model that defines components and composition mechanisms that can be used in these life cycles. To this end, we use the X-MAN component model [10], [5], [8].

B. The X-MAN Component Model

In the X-MAN component model (Fig. 3), there are two basic entities: (i) *components*, and (ii) *composition connectors*. Components are units of design with behaviour; behaviour is exposed through a component's interface as provided services. Composition connectors are composition mechanisms for components.

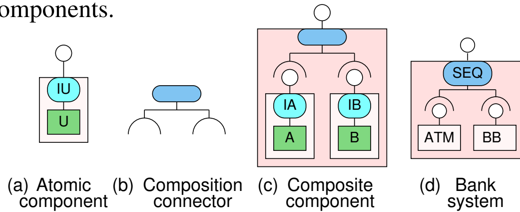

Fig. 3. The X-MAN component model.

Components can be atomic or composite. An atomic component (Fig. 3(a)), is formed by composing an invocation connector (*IU*) with a computation unit (*U*). The computation unit contains an implementation of some behaviour (methods) in a chosen programming language, e.g. C. More importantly, the computation unit must have complete behaviour, i.e. it is *not* permitted to call another computation unit (of another component) in order to complete its behaviour. The invocation connector exposes the methods of the computation unit to the component's interface and allows them to be executed.

Composition connectors (Fig. 3(b)) are control structures. They are used to compose (atomic or composite) components into composite components. In a composite component (Fig. 3(c)), sub-components do not call each other. Instead, the composition connector coordinates the sub-components' execution.

Two basic composition connectors are *Sequencer* and *Selector*. *Sequencer* provides sequencing, whilst *Selector* offers branching. For instance in the bank system in Fig. 3(d), the sequencer *SEQ* first calls *ATM* to get customer inputs and then calls the bank branch *BB* to handle the inputs.

In addition, there are special connectors that are not used for composition, but for adapting single components. These connectors are called *adaptors*; examples are *Loop* and *Guard*. *Loop* allows for iteration (over a single component) and *Guard* offers gating.

The execution semantics for X-MAN is control-driven. Component execution is initiated and coordinated by (composition) connectors. Connectors initiate control to invoke components. At each component, control then triggers read and write operations on data channels associated with the component in order to supply inputs to the component, and distribute the outputs from the component. Data routing can be *horizontal* or *vertical*. *Horizontal* data routing is between sub-components within a composite component. *Vertical* data routing is data propagation between the interface of a composite component and its sub-components. A data channel has a capacity of 1, and can have two possible read policies: *destructive* and *non-destructive*. Moreover, a data channel can be initialised to contain an initial value.

C. Using X-MAN Components with the W Model

X-MAN components can be used for system development following the W model. This is because X-MAN components have a life cycle that matches the W model. This life cycle has 2 phases: (i) component design (and implementation), and (ii) component deployment. Component design is concerned with building repository components and therefore coincides with the component life cycle of the W model. Component deployment uses repository components to construct systems, and therefore coincides with the system life cycle of the W model.

In the component design phase, X-MAN components are templates that are meant to be suitable for a number of systems (in a domain). Hence, the repository contains templates. In the component deployment phase, X-MAN components are instances of components in the repository. Instances are created by instantiating, adapting and initialising templates. Because of that, instances tend to be specific to the system they are part of. Therefore, it is not desirable to store these instances in the repository.

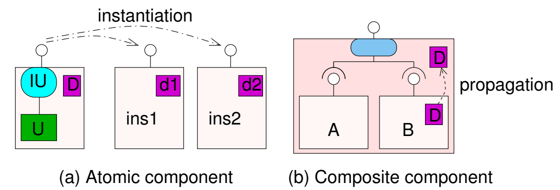

Fig. 4. Initialisation data in X-MAN component.

To serve the purpose of component instantiation and initialisation, a component in the component design phase can declare its required initialisation data in its interface. An initialisation data declaration contains a type, a default value, and optionally a value range. Initialisation data of a composite component are propagated from its sub-components, i.e. by

vertical data routing. An example is shown in Fig. 4. In Fig. 4(a), the atomic component declares a data element *D* which is then initialised to *d1* and *d2* in two instances, *ins1* and *ins2*. In Fig. 4(b), the data element *D* of the sub-component *B* is propagated to the composite component.

In addition, during component deployment, component adaptation allows us to enable only selected services out of all the provided services of a component. For example, in Fig.3(d), although the bank branch component in the repository may have three services, namely *Withdraw*, *Deposit*, *Balance*, the deployed instance *BB* may have only the *Withdraw* and *Balance* services.

III. The X-MAN Tool

The X-MAN tool supports component-based development using X-MAN components and following the W model. It is implemented in a model-driven manner using an MDE tool called GME (Generic Modelling Environment).

A. The Generic Modelling Environment (GME)

GME [14] is a generic and customisable modelling environment. As comparable to the de facto standard GMF, GME customisation begins with metamodelling. A metamodel in GME is essentially a class diagram with well-known concepts such as inheritance and containment. In addition, there are notions of proxy and references. Proxy allows for cloning a metamodel element so that clones could co-exist and be arranged conveniently in a large and complex metamodel. This in effect enables us to define clear and well-organised diagrams. Reference allows for creating references to metamodel elements.

Once a metamodel is created, it needs to be registered with GME. After that, models can be instantiated from the registered metamodel.

An important feature of GME is the concept of *aspect*. An aspect defines a view in which specified metamodel elements are displayed. In general, there could be overlapping between aspects if the shared metamodel elements are used to link the aspects semantically. Aspects are part of a metamodel in GME and therefore, they will be registered to GME as a part of metamodel registration.

Interpreter is another significant concept in GME. An interpreter can take an active model and process it. Model processing can vary from traversing to inserting and modifying model elements. Traversing a model is perhaps the most interesting as it enables us to perform model validation, transformation and execution. Obviously, an interpreter needs a concrete implementation to determine how the model interpretation is done.

In order to define an interpreter, GME provides a wizard embedded in Microsoft Visual Studio. The wizard asks for the interpreter name and metamodels on which the interpreter will be working. An interpreter can register to any number of metamodels or all available ones.

Like GMF, GME generates C++ classes for metamodel elements. Accessing models is by calling accessor methods of these classes in an interpreter implementation. Moreover, it is encouraged to use the GoF *Visitor* design pattern [4] in implementing interpreters. Each metamodel class is generated to contain the *accept* method and an extra Visitor class is also generated to have a number of *visit* methods for all metamodel classes. Through implementing the *visit* methods, we can access any model elements for any model processing mentioned above.

B. Implementation Outline

Following GME principles, we developed our metamodels and interpreters. Specifically, we defined two metamodels: one for the component life cycle and one for the system life cycle.

Having the metamodels immediately makes the modelling functionality available. However, in order to fully implement the activities in the life cycles, we had to add extra implementation in the form of interpreters. We developed two sets of interpreters that are called *Component Designer* and *System Assembler*.

We used C++ to implement all our interpreters. In addition, MySQL [11] was used as the database engine to host the repository and an ORM framework called ODB [3] was utilised for rapid development. In order to execute component implementation in C, we employed a C interpreter called Ch [13]. We also linked into GME an external code editor called SciTE [12]. SciTE could be configured to call a suitable compiler/interpreter, which is Ch in our case, to compile source codes.

C. Component Life Cycle Metamodel

The metamodel for the component life cycle is presented in Fig. 5. The top-level element is *AtomicComponent* and *CompositeComponent*, which contains other elements to construct the the definition of a (atomic or composite) component. *ComputationUnit* has an attribute, namely *ExecutableCode*, to capture an implementation. *CompositeComponent* contains one or more *Connector* and a number of *Component*, which are design phase instances. *Connector* can be *Sequencer*, *Selector*, *Loop* and *Guard*. More importantly, components have a number of *Service* with *Input* and *Output*. Component could also have *DataElement*, which models initialisation data. Moreover, there is *DataChannel* that connects a pair of parameters, which can be inputs and outputs. Like *DataChannel*, *DataAssignment* links two data elements to allow for propagation of data initialisation.

Optionally, *Contract*, which is formally defined as a pair of pre- and post-condition, can be defined for services of components for verification purposes. Components have the attribute *Verified* to indicate whether all contracts have been verified successfully.

For the separation of control and data, we defined two aspects, which are *ControlView* and *DataRoutingView*. *ControlView* is set to show components, services and connectors, whereas *DataRoutingView* is assigned to display services, initialisation data and data channels. *DataRoutingView* also displays *Selector*, *Loop*, and *Guard* as they own inputs.

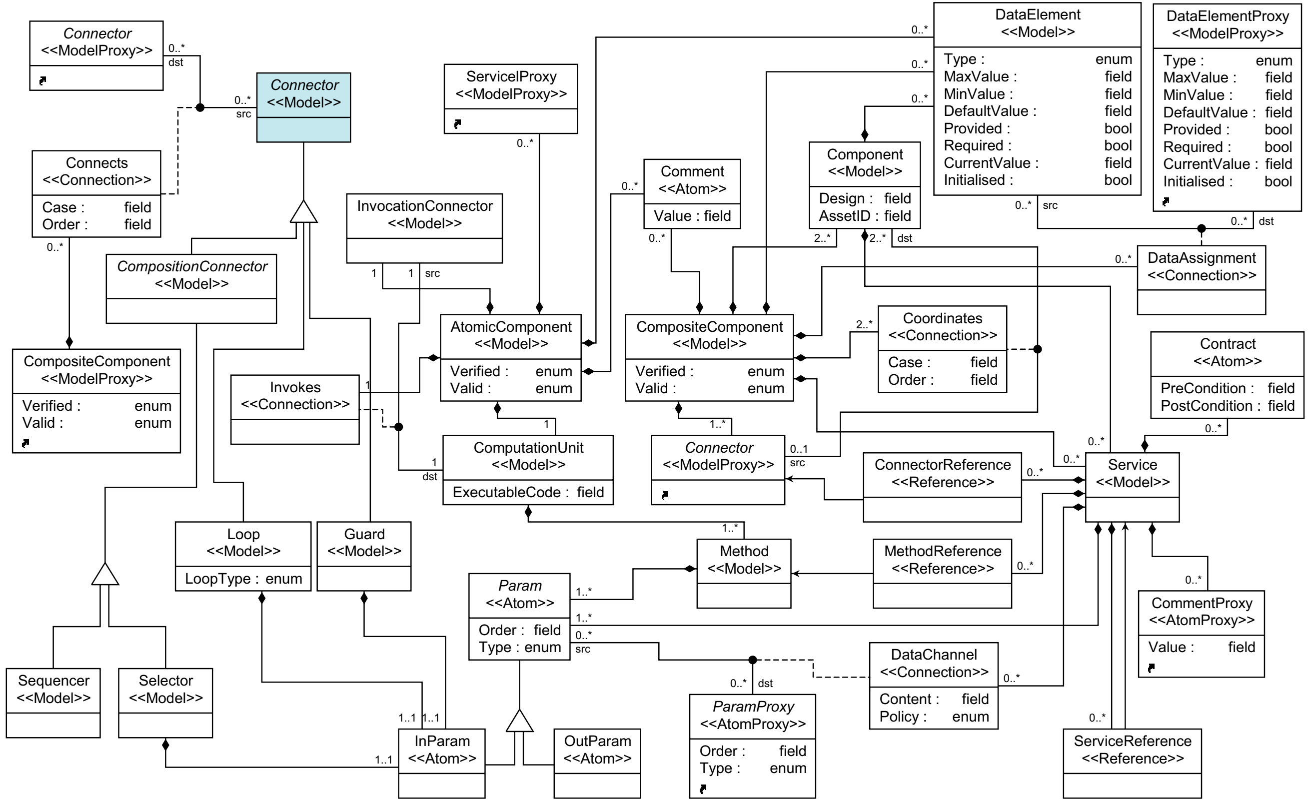

Fig. 5. Metamodel of component life cycle.

D. Component Designer

Component Designer consists of a number of interpreters that supports component design, validation, verification, deposit and deployment. They appear as *IGn*, *VAL*, *VER*, *DEP*, *RM* and *RET* tools in a toolbar as depicted in Fig.6.

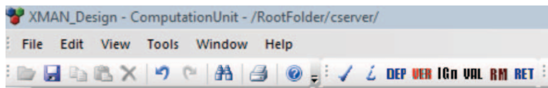

Fig. 6. Component life cycle toolbar.

1) Component Design: For an atomic component, the design environment is presented in Fig.7. In this instance, an atomic component called *CllVoter* was designed. There is a design palette on the left, a tree view of the design on the right, and the main design view in the middle. The main view shows that *CllVoter* is an atomic component built by connecting an invocation connector to a computation unit. The computation unit has its implementation in the bottom left text box. The source code is in C and could be annotated with pre-defined tags to support automatic component interface generation. Interface generation is triggered by pressing the *IGn* button.

The component has its interface containing the provided service *vote* (purple box), which takes five inputs (blue boxes) and produces one output (green box).

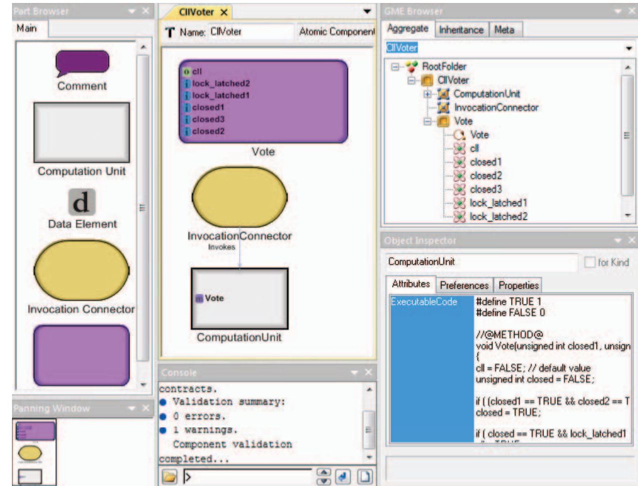

Fig. 7. Designing an atomic component.

For composite component design as visualised in Fig.8, the design palette is updated to display suitable connectors for building composite components. These connectors are *Selector*, *Sequencer*, *Loop*, and *Guard*. There are also other design entities such as *Service* and *DataElement*. Composite components are built by composing component instances. In our example, an instance of Sequencer connector called

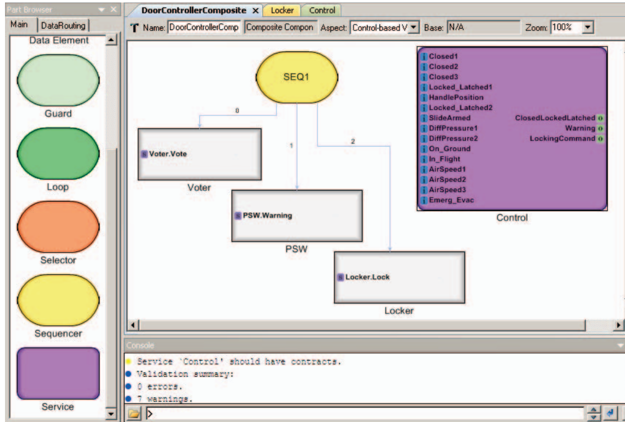

Fig. 8. Designing a composite component (control view).

SEQ1 composed three component instances *Voter*, *PSW* and *Locker*. Note that *Voter* is an instance of the previously constructed atomic component *CllVoter*. Component execution sequence is specified on the connections between *SEQ1* and the instances e.g. 0,1, and 2. Like atomic components, the composite component has its interface consisting of inputs and outputs. For instance, the service *Control* has fourteen inputs and three outputs.

Having defined the control in the composite component, we also had to specify the data routings, i.e. both vertical and horizontal ones. Fig.9 shows our data routings for this composite component. The horizontal data routing consists of

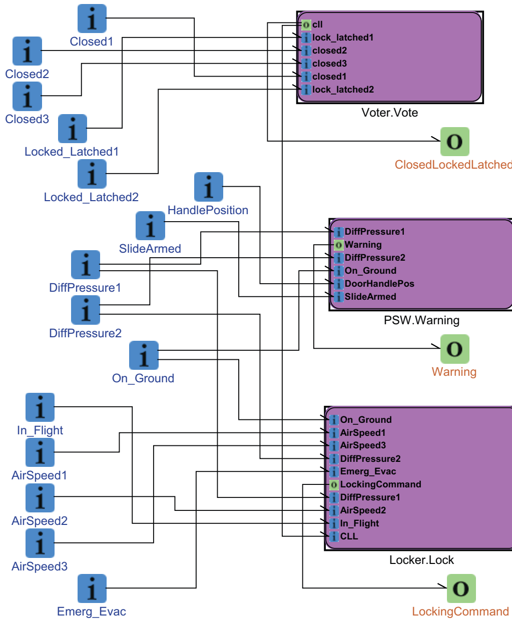

Fig. 9. Designing a composite component (data routing view).

the only data channel connecting the output *cll* of *Voter* to the input *CLL* of *Locker*. All other data channels contribute to the vertical data routing.

As can be easily seen, we have modelled control and data separately. This gives clarity and expressiveness. However, we need to ensure that the models (e.g. Fig.8 and Fig.9) are not contradicting. Hence, we implemented a validation tool called *VAL* for this purpose. The tool uses the control view as the base model and traverses the data view checking data channel directions and channel ends. For instance, the channel connecting inputs *Closed1* and *closed1* (of Voter.vote) is valid because it defines a valid vertical data flow from the composite component interface to the sub-component Voter's interface. In addition, the channel flowing from output *cll* (of Voter.vote) to input *CLL* (of Locker.Lock) is also valid because it complies with the execution sequence, i.e. Voter first and then Locker.

2) Component Retrieval and Deposit: As mentioned before, composite components are built from component instances. Component instantiation starts by pressing the *RET* tool. The tool provides a chain of component selection and instantiation dialogues. We need to provide the necessary details, which are instance name, services and initial values, to create component instances. The component instantiation dialogue is visualised in Fig.10. In this example, we specified the instance name as *Voter* and selected the provided service *Vote* to have in the instance. The latter step is important because components can have many services but we need to be able to adapt an instance to have enough functionalities, in order to construct the required functionalities of a composite component.

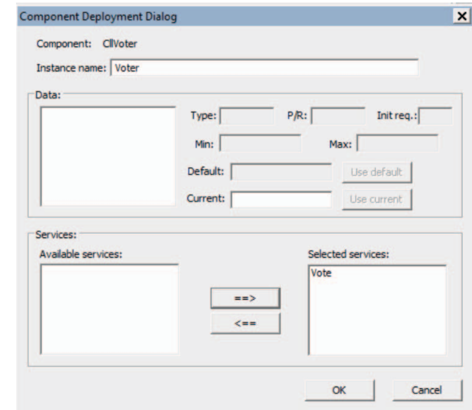

Fig. 10. Instantiating a component.

Component deposit is triggered by running the *DEP* tool. When executed, component deposit automatically triggers the component validation tool to guarantee design validity. If the design is valid, it will be stored in the repository.

3) Repository: Repository management is implemented via the *RM* tool. *RM* offers repository management with facilities such as component lookup, removal, sorting, exporting and categorisation. We offered these facilities through a set of dialogues triggered in the *RM* tool. The main dialogue is shown in Fig.11. There is a list of components on the left and a component summary view on the right. Component list can be shorted by using the filtering tool at the top. In addition, components can be removed and categorised using the repository management tools at the bottom. Finally, component interface can be revealed in order to support component selection.

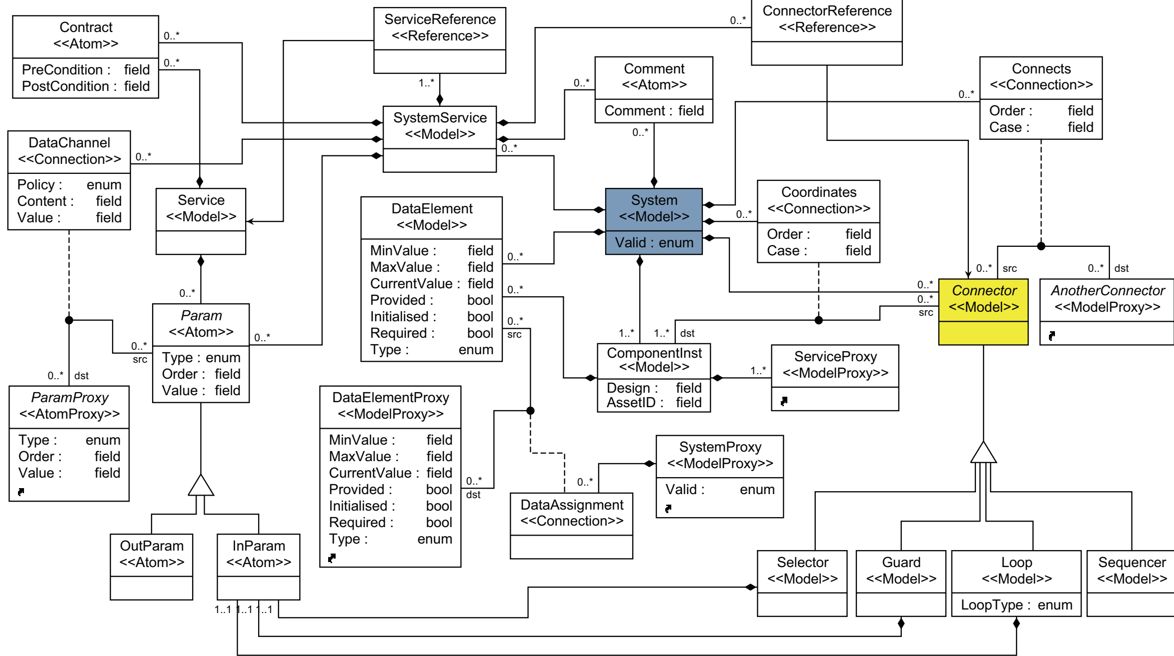

Fig. 12. Metamodel of system life cycle.

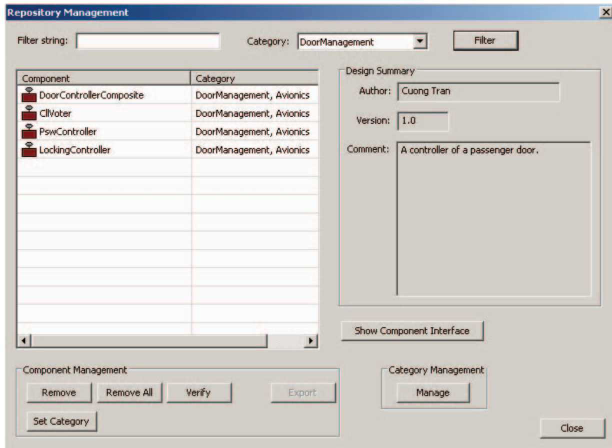

Fig. 11. Repository.

E. System Life Cycle Metamodel

The metamodel for system life cycle is presented in Fig. 12. In this life cycle, *System* is the top element containing all others, as opposed to (atomic and composite) components in the component life cycle. A system is formed from a number of *ComponentInst* and *Connector*. Connectors are the same ones in the component life cycle. Similarly, there is *DataChannel* that connects a pair of parameters, which can be inputs and outputs. System can have a number of *SystemService*. Each *SystemService* has *Input*s and *Output*s and necessary *DataChannel*s. Note that, *SystemService* can also contain *Contract*, which is also defined as a pair of pre- and post-condition. Contracts of a system service are supposed to be generated from and verified against those of components by using a suitable verification approach.

As in the component life cycle, we defined two aspects to separate control and data modelling in systems.

F. System Assembler

As shown in Fig.13, *System Assembler* consists of a set of tools for system design, validation, verification and simulation. These tools are implemented as *RET*, *VAL*, *VER* and *SIM* on the toolbar.

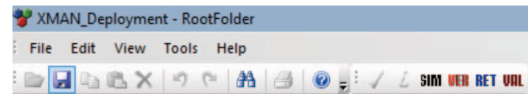

Fig. 13. System life cycle toolbar.

1) System Design and Validation: Like composite component design, the system design environment has a design palette that offers connectors, data element and system interface. A system is constructed by composing component instances with connectors. The *RET* tool is used for component instantiation and initialisation. Component instantiation in this life cycle requires model transformation, i.e. from Atomic-Component and CompositeComponent in metamodel in Fig.5 to ComponentInst in Fig.12. The transformation is rather straightforward because the key elements, which are Service and DataElement, in both metamodels are conveniently the same.

An example of system design is depicted in Fig.14. Two instances namely *Door1* and *Door2* are composed by a connector *SEQ1*. Again, the execution sequence of the two instances is specified on the connections, e.g. 0 and 1. The

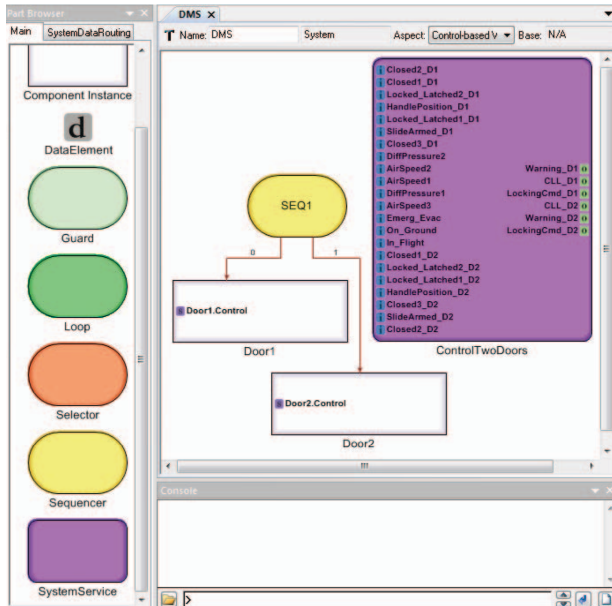

Fig. 14. Designing a system (control view).

system interface offers a number of inputs and outputs that are built from those of the instances.

As well as the control view, a data view needs to be created for the system. We omit this view for lack of space.

System design validity is checked by launching the *VAL* tool which works in the same way as it does on composite components. The tool goes through all data channels and makes sure that they are consistent with the control in the control view.

2) Simulation: In order to validate a system assembly against its requirements, we can define a number of test cases with inputs and assertions, and use our *SIM* simulation interpreter to execute the system assembly against these test cases.

Once the system is fully defined, we validated the system by simulating it and evaluating its outputs using a set of test cases derived from the system requirements. The simulation tool *SIM* is visualised in Fig.15. The simulator is showing the signature of the selected system service. The service requires a number of inputs such as *DiffPressure1*, *DiffPressure2*, *On_Ground* and so on.

The simulation will iterate once through all test cases and log inputs and outputs at every possible elements. When the simulation is finished, the results are displayed in a separate dialogue. We can select an individual test case and inspect the execution traces to debug the system. The simulation result dialogue is depicted in Fig.16.

3) Verification: Component and system verification are also implemented as interpreters, which are named *VER* in both Fig.6 and Fig.13. They simply export component and system designs in a suitable format, for instance XML, and pass them to suitable verification tools, such as CBMC and XMAN Verifier as in the work of [5].

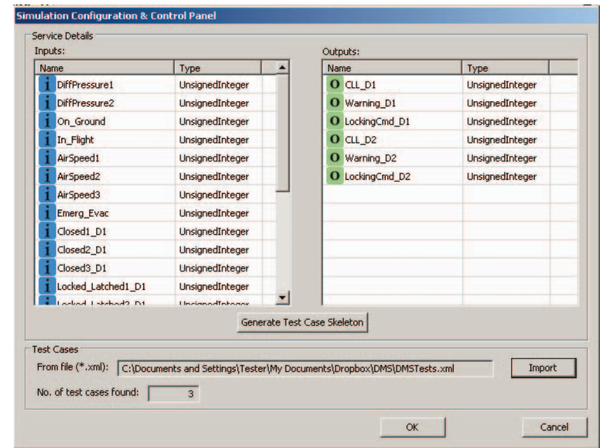

Fig. 15. The simulator.

Fig. 16. A simulator result.

IV. An Industrial Example

Now we present an industrial example from the avionics domain that we have implemented using the X-MAN Tool. The example is a simple version of the Door Management System (DMS) on a civil aircraft;[2] it manages only two passenger doors (the most complicated system can consist of up to fourteen doors).

Structurally, each door is equipped with a number of sensors that produce readings of closed status, lock-latched status, door handle position, and slide armed status. In addition, there are inputs coming from other systems on the aircraft (e.g. differential pressure, air speed readings).

Functionally, the system monitors the door status, manages the evacuation slide, and controls the door lock actuator for each door. The door status is calculated from closed, locked, and latched signals.Also, a warning is activated when the inner door handle is moved and the evacuation slide is in armed position. The evacuation slide management furthermore generates a warning if the aircraft is on ground, the cabin is still pressurised but the slide is disarmed. Finally, the door lock

[2]This is a pilot application on the CESAR project.

control only permits the door to be unlocked if the aircraft is on ground and the situation is 'emergency evacuation' or windspeed is lower than 12kph.

From our analysis of the DMS system, we identified three components namely CllVoter, PSWController, and LockingController. These components are common and can be used many times in possible variants of DMS, e.g. a DMS variant consisting of 4 passenger doors. Hence, in the component life cycle, we designed the three components and deposited them in the component repository.

To realise the above functionality, we needed to compose the identified components. Moreover, since all doors have the same functionality, it is more efficient to build the functionality as a composite component so that every door is an instance of this composite component.

Therefore, after creating the three above atomic components, we create three instances to build the composite component *DoorControllerComposite*, as depicted in Fig.8.

Then, we implemented the DMS system by instantiating the composite component *DoorControllerComposite* twice and composing these instances as in Fig.14 [3].

We validated that system by defining three test cases and testing the system against them. The results (in Fig.16) told us that the system behaves exactly as expected. In particular, the first test case expects the outputs *LockingCmd_D1* and *LockingCmd_D2* to be 1. The execution trace shows exactly that the two actual outputs (at the bottom) are also 1.

V. Evaluation and Conclusion

Compared to the initial version in [10], the X-MAN tool in this paper has evolved tremendously. The successful evaluation by Airbus Operations Limited for the avionics domain not only resulted in many alterations and improvements, but it also highlighted the major industrial requirements that X-MAN now meets. One such requirement is the construction of composite components in the component life cycle. Composite components allow for reusing 'big' components, which leads to simpler but highly factored designs. Another industrial requirement is hierarchical component and system construction. Hierarchical construction leads to compositionality, which in turn tackles scale and complexity. It is thus crucial for systematic construction of large complex systems. Clear data routing is another requirement, by Airbus engineers. For this we introduced data channels so that control and data can be modelled independently. Such separation is significant for effective modelling as well as architecture analysis and reasoning.

The MDE approach has enabled us to realise all these improvements relatively quickly and painlessly.

In a wider scope, tools for component-based software development generally support ADL-like component models in which components are architectural units. More importantly, components are usually products of system analysis and design, i.e. there is no true component life cycle or well-defined repository. Components are normally abstract entities and hence have no implementation, with the exception of Modelica [1] and SOFA2 [2]. Moreover, modelling in these tools does not offer clear separation of control and data.

[3]Ideally, we could have used a better connector called *Cobegin* that provides concurrency [7]

In contrast, the X-MAN tool supports the W model. Components are always defined for reuse (for a domain) and can also be pre-validated. Systems always use components from a well-defined repository to ensure quality and cost savings. In addition to that, the tool encourages separation of concerns in development by clearly distinguishing the two key concepts in modelling, namely control and data. It also emphasises separate development phases for component design and deployment. As a result, roles, such as component developer and system assembler, can be clearly defined and applied in development for efficient management.

In the future, more changes are needed. We need to implement a code generator that is capable of generating a restrictive set of code complying to the avionics DO-178B standard. To this end, we intend to utilise the model transformation tool suite called GReAT [6]. Also, we will seek opportunities to apply the X-MAN tool to other domains. Finally, we want to improve the V&V aspect in order to support more properties such as safety and reliability.

References

[1] Modelica Association. Modelica tools https://www.modelica.org/tools.

[2] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[3] CODE SYNTHESIS TOOLS CC. Odb: C++ object-relational mapping (orm) http://www.codesynthesis.com/products/odb/.

[4] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley, 1995.

[5] N. He, D. Kroenig, T. Wahl, K.-K. Lau, C.M. Tran, F. Taweel, P. Rummer, and S. Sharma. Component-based design and verification in x-man. In *To appear in Proceedings of ERTS 2012*, 2012.

[6] Vanderbilt University Institute for Software Integrated Systems. The graph rewrite and transformation (great) tool suite http://repo.isis.vanderbilt.edu/tools/get_tool?GReAT.

[7] K.-K. Lau and I. Ntalamagkas. Component-based construction of concurrent systems with active components. In *Proc. 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2009)*, pages 497–502. IEEE, 2009.

[8] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In G. Heineman *et al.*, editor, Proc. 8th Int. Symp. on Component-based Software Engineering, LNCS 3489. Springer, 2005.

[9] K.-K. Lau and Z. Wang. Software component models. In *IEEE Transactions on Software Engineering*, volume 33. IEEE Computer Society, October 2007.

[10] Kung-Kiu Lau, Faris M. Taweel, and Cuong M. Tran. The w model for component-based software development. In *Proceedings of SEAA 2011*, Finland, 2011. IEEE Computer Society.

[11] Oracle. Mysql:: the world most popular open source database http://www.mysql.com/.

[12] Scintilla project. Scite - a free source code editor for win32 and x http://www.scintilla.org/, 2012.

[13] SoftIntergration. Ch – an embeddable c/c++ interpreter, c and c++ scripting language http://www.softintegration.com/.

[14] Vanderbilt University. Gme: Generic modeling environment http://www.isis.vanderbilt.edu/Projects/gme/, 2008.