Automatic Generation of Executable Code from Software Architecture Models

Aristos Stavrou[1] and George A. Papadopoulos[1]

[1] Department of Computer Science, University of Cyprus, {cs98sa2,george}@cs.ucy.ac.cy

Abstract. Our effort is focused on bridging the gap between software design and implementation of component-based systems using software architectures at the modeling/design level and the coordination paradigm at the implementation level. We base our work on the clear support of both software architectures and event-driven coordination models for Component Based Software Engineering and the similarities we have identified between the fundamental concepts of software architectures and the event-driven coordination model. Exploiting the improvements realized by the latest version of UML towards the support of software architecture descriptions, we present a methodology for automating the transition from software architecture design of component-based systems described in UML 2.0 to coordination code. The presented methodology is further enhanced with a code generation tool that fully automates the production of the complete code implementing the coordination-communication part of software systems modeled with UML 2.0.

1 Introduction

Our effort is focused on bridging the gap between software design and implementation of component-based systems using software architectures at the modeling/design level and the coordination paradigm at the implementation level. Our choice was based on the clear support of both software architectures and event-driven coordination models for Component Based Software Engineering and the similarities we have identified between the fundamental concepts of software architectures and the event-driven coordination model.

In (Papadopoulos, Stavrou and Papapetrou 2006) we have presented a methodology for mapping ACME (Garlan, Monroe. and Wile 2000), a generic language for describing software architectures, down to event-driven coordination code in the Manifold (Arbab, Herman and Spilling 1993; Papadopoulos and Arbab 2001) language. The reason for using ACME was precisely in order to show the generality of our approach: since ACME embodies the core features that any state-of-the-art Architecture Description Language (ADL) would support, by mapping ACME to Mani-

fold we effectively provide the core of an implementation route for any other ADL (Medvidovic and Taylor 2000).

Based on the results and experience of our first work and exploiting the improvements realized by the latest version of UML towards the support of software architecture descriptions, we propose a new methodology for modeling the software architecture of a component based system in UML 2.0 (OMG 2003) and the automatic transition of this model to event-driven coordination code in Manifold. Our latest work targets an improved support for the dynamic aspects of the software architecture exploiting the powerful tools of UML for dynamic behavior. Furthermore, we use the standards (UML2.0, XMI) and approach proposed by the new software development discipline which promises to be the next big step in software development, similar to the move from machine language to compilers forty years ago, namely the Model Driven Architectures (OMG MDA Website 2006). The presented methodology is further supported by a code generation tool that fully automates the production of the complete code implementing the coordination-communication part of software systems modeled with UML 2.0.

2 From UML 2.0 Software Architectures to Coordination Code

Our methodology addresses the challenging task of automatic code generation. The task of automating the translation of a system high-level model described in a general purpose modeling language, such as UML, to executable code has nothing to do with magic, but with:

1. The methodical construction of this high level model using the proper constructs provided by the modeling language having in mind the target programming model. This means that the modeler (in our case the software architect) should be guided on the way that s/he will use the modeling language according to the desired goal. Modeling languages such as UML are not associated with a specific methodology but they just provide the constructs and notation to model systems of different domains. Our models must adequately describe all aspects of a system's software architecture and provide all necessary detail to our code generation tool in order to produce complete and functional code.
2. The accurate and clear mapping of these constructs to the lower-level constructs used by the target programming model.

Satisfying the first requirement, our methodology defines, at first stage, the constructs to be used to model both static and dynamic aspects of a component-based system's software architecture and then guides the software architect on how to use these constructs.

The static aspects of a component-based system's architecture include the definition of the components (and sub-components) that the system is composed of, the functionality provided and required by each of them as well as the definition of all possible interactions realized between components to accomplish the tasks that the system is supposed to implement. In UML 2.0 these are adequately described by the constructs provided for architecture modeling such as components, classes, ports,

interfaces and connectors. Architecture modeling in UML 2.0 is realized by component diagrams (also called architecture diagrams).

The dynamic aspects of a component-based system's architecture include the setup of the software architecture under certain execution scenarios. This includes the subset of interactions taking place between components, realized by messages or events exchanged between them. The dynamic aspects may also include the activation or deactivation of component instances (and subsequently the new setup of interactions between them) in response to changing factors of the system's execution environment such as the load of the system at a specific time. The order in which the above actions may take place in response to the dynamic behavior of the system is very important and it has to be precisely defined in our model in order to produce the corresponding coordination code. The constructs provided by UML 2.0 for scenario (or interaction) modeling can adequately describe the above actions and the specific order in which they might occur.

From diagrams provided by UML 2.0 for scenario modeling, we choose sequence diagrams as the most suitable for precise and detailed description of a system's actions and behavior.

Sequence diagrams are used to define:

- the messages and events exchanged between specific component and class instances under certain scenarios,
- the details of other actions that may occur in a system such as activation/deactivation of component and class instances,
- the conditions under which such actions might take place,
- the specific sequence in which these actions will take place.

The two types of diagrams that are used in our methodology are perfectly interrelated, thanks to the new feature of UML 2.0 for structure and behavior gross integration. This means that the model elements specified in the architecture diagrams such as components, classes, connectors, and interfaces are then directly associated with the model elements included in sequence diagrams. For example, each lifeline of sequence diagrams is directly associated with a component or class defined in architecture modeling, and the messages associated with this lifeline realize the interactions that have been specified for this component or class in architecture diagrams through connectors. The "call" type messages included in sequence diagrams invoke operations defined to be provided or required by components through interfaces and ports in architecture modeling. This useful feature of UML 2.0 provides us with the ability for consistency checking between elements specified in different diagrams but also makes the process of automatic translation of diagrams to programming constructs easier since we do not have to proprietary define the relationship between the model elements of the different diagrams that we will use. However, although the specific feature is provided by UML 2.0, this does not mean that it will be used by the software architects. We give general guidelines to software architects for the way that they use this feature during modeling in order to take advantage of it. Furthermore, we define constraints in the way that they will use the different constructs of diagrams.

Satisfying the second requirement mentioned above, our methodology defines an accurate and clear mapping of the higher-level constructs of UML 2.0 to lower-level

constructs provided by our programming model. Our construct (or concept) mapping is aided by the precise and unambiguous semantic definitions of UML 2.0 (an improvement made to satisfy one of the main requirements for Model Driven Development) and the explicit relationship realized between the constructs of UML 2.0 and Manifold, the formal representative of control-driven coordination model that will form the target programming language of our methodology. Unlike in our previous methodology (Papadopoulos et al. 2006) where we had to define events, interaction ordering and all other aspects relating to the dynamic behavior of the system in the form of proprietary properties, UML 2.0 does not only provide us with first class constructs that explicitly model these aspects but the semantic meaning of these constructs, as this is defined in UML 2.0 superstructure (OMG 2003), is very similar (in most cases identical) to the constructs realized by control driven coordination models and Manifold for implementing the same aspects. This explicit mapping of high-level model element to control driven coordination elements enable us to extract specific coordination constructs from each element included in our model diagrams, rather than proprietary extracting specific programming code parts out of specific pieces of a modeling language notation.

In the following sections we present an overview of the steps defined by our methodology to create the architecture and scenario model of a component-based system in UML 2.0 as well as the general rules followed to map these models to coordination code in Manifold. Due to space limitations, a detailed description of the steps and mapping rules of our methodology cannot be included in this paper and will appear in an extended version of it for a journal submission.

2.1 Creating the Architecture Model of a System's Architecture

Architecture modeling is realized by a number of component (or architecture) diagrams describing the static aspects of a component-based system's architecture, which include:

- the definition of all different (types of) components, sub-components and classes that the system is composed of ,
- the functionality provided and required by each of them realized by provided and required interfaces of components and classes,
- all possible interactions between components and classes realized as connections between ports of them.

For the construction of architecture diagrams the UML 2.0 constructs that are used by our methodology are: *components, classes, ports, interfaces (operations, signals, attributes) and assembly/delegation connectors*.

Apart from the clear semantics given by UML 2.0 for each construct of architecture modeling, the software architect has to be guided on the way that he will use these constructs to create appropriate architecture diagrams that will adequately describe the required information of the system under development and enable the automatic transition of these diagrams to the target programming model which is Manifold.

Our methodology satisfies this need by defining a number of constraints on the use of the UML 2.0 architecture constructs. Some of the constraints defined by our methodology on the use of the above constructs are the following ones:

- Only one operation per interface is allowed. This helps us to clearly define the connections between ports that interfaces are attached to and internal parts that implement the functionality defined by this interface.
- Each port must be attached to either a required or a provided interface. This enables us to easily and explicitly model the relationships between the ports and the internal parts of the component.
- Both a name and type must be defined for each attribute of an interface.

A number of general guidelines on the way that architecture diagrams will be constructed are also defined. Generally speaking, the software architect will have to follow the principles of component based development to identify the components, sub-components and classes of the system and describe the functionality provided and required by them, but also the target programming model has to be considered. The general steps for the construction of the diagrams are the following ones:

1. Identify the top-level components of the system architecture. Create a top level diagram and add a special *Main* component. (This will represent the special manifold process that every system in manifold should include). Add the top-level components of the system as sub-components of the Main Component.
2. For each component identify the different operations that are provided by this component.
3. For each operation identify the different parameters that are needed to be given to the component to execute this operation and the possible values returned by this operation. Create an interface for each operation and add the specific operation with its parameters and return values.
4. Identify the possible signals sent by the component providing this operation to its environment in response to a call on this function. Add the signals to the created interface.
5. Identify possible main variables related to the operation that can be identified at this stage and may affect the setup of the architecture. E.g. a variable with name “requests_rate” that counts the number of requests per minute that is used by the component to instantiate a new instance of a sub-component or class if the rate exceeds a certain number. Add these attributes to the created interface.
6. Identify the required operations and create an interface for each of them in a similar way as above. For each required interface add a signal sent by the component requesting the call of the related operation to its parent component that coordinates it in order to create the needed setups (connections).
7. For each component add a port for each provided/required operation of the component and attach it to the corresponding required or provided interface.
8. Identify all possible connections between the sub-components of a first level component. Create an assembly connector for each connection between the ports that the related provided and required interfaces are attached to.

9. Identify all possible connections from the top level component to its parts (sub-components, classes). Create a delegation connector for each such connection.
10. Decompose each of the sub-components to another diagram. Add in the new diagram the specific sub-component as the top level component and add all sub-components and classes that this component is composed of. Follow the steps above to add the ports, required/provided interfaces as well as the connections.

2.2 Mapping an Architecture Model to Equivalent Manifold Constructs

The translation of the architecture model constructs will give as the manifold coordinator (manager) and atomic (worker) processes that the system will be composed of, the input and output ports of each process as well as the possible events that are raised by each process to its environment. Furthermore, all possible streams created between ports of system processes will be extracted from the architecture model. Finally some local variables used by processes for specific operations as well as extra control ports and guards installed on these ports to notify the receipt of an operation call will be created. More to the point:

A *Component* can be exactly mapped to a *Manifold coordinator process*. An *Active Class* is mapped to a *Manifold atomic process*. *Passive Classes* will be used in our architecture modeling to represent the different data types supported by Manifold such as *string, integer, tuple,* etc.

An *interface* is not directly mapped to a specific Manifold construct but the set of operations, attributes and signals defined for the specific interface are separately mapped. For every *operation* that is defined in a provided or required interface attached to a port, we create an *input port* for each input parameter and an *output port* if the specific operation returns a value. A special *input control port* is also created for each operation and a *guard* is installed on this port to notify the owning manifold process for requests received for the specific operation. The set of *signals* defined in provided and required interfaces attached to a component's or class' ports are defined to be the *events* that can be raised by the corresponding manifold or atomic process. *Attributes* of an interface attached to a component or class are mapped to *local variables* of the corresponding Manifold coordinator or atomic processes. The variables will be assigned an initial value if the corresponding interface attribute is assigned a value.

Connectors are mapped to the streams required to pass the related parameters values during an operation call and streams to pass the return values of an operation.

2.3 Creating the Scenario Model of a Software Architecture

Scenario modeling is realized by a number of sequence diagrams describing the dynamic aspects of a component-based system's architecture, i.e. the setup of the software architecture under certain execution scenarios. Specifically, sequence diagrams describe:

- the interactions taking place between components, realized by messages exchanged between them,
- the activation/deactivation of component and class instances,
- the conditions under which the above actions take place,
- the sequence within which the above actions take place.

For the construction of sequence diagrams the following constructs provided by UML 2.0 will be used by our methodology: *lifelines, messages (operation calls, signals, create, destroy, display, set), timers, inline frames, message groups and gates*.

The goal of the software architect during the creation of the sequence diagrams is to describe all possible execution scenarios of the system. As soon as the different scenarios are identified, the software architect can create in a hierarchical top-down approach the sequence diagrams of each scenario as follows:

1. Create a top level sequence diagram and include a lifeline for the "`Main`" component and a lifeline for each instance of the first level components/classes that are involved in the execution of the first execution scenario.
2. Use the constructs for scenario modeling described above to define the interactions – messages taken place during the execution of the first execution scenario. Bear in mind that this sequence diagram will describe the interactions-actions from the perspective of the parent component, i.e. this sequence diagram will produce the events received by a parent component from its parts and the actions that the parent component has to make during the execution of the current scenario for coordinating its parts.
3. Decompose every decomposable lifeline to another sequence diagram, describing the message exchanges taking place for the current scenario at a lower lever (i.e. between the specific component and its part's instances). In the new sequence diagram, add a lifeline representing the component that is decomposed (i.e. the parent component) and then a lifeline for each instance of the component's parts.
4. Add all "signal" and incoming "operation" call messages of the higher level sequence diagram that are attached to the lifeline currently being decomposed.
5. Between the already created messages, add all message exchanges taking place between the decomposed lifeline (i.e. the parent component) and the other lifelines.
6. For each component, create a new sequence diagram with a special name "Component name - Init" in order to describe the initialization process of the component such as the creation of process instances. The same diagram can include any finalization process that may exist or any other scenarios that has not been described in diagrams before, e.g. the activation of a new component/class instance when the load of the system is high.

2.4 Mapping a Scenario Model to Equivalent Manifold Constructs

A scenario model describes the dynamic aspects of the system architecture. In the "world" of Manifold this is translated to a number of states for each component and a number of actions (such as the creation of new instances, the creation of streams,

raising of events) executed by the specific component when it is in this state. Mapping of a scenario model is performed after the mapping of architecture diagrams since the mapping of a scenario modeling builds upon (uses) the Manifold constructs extracted while mapping the architecture model. In the architecture model we have extracted the definitions of manifolds and atomics (i.e. names, events raised by them to their environment, ports, local variables) of our system architecture and the hierarchy between them, i.e. which manifold coordinates which other manifolds or atomics. We have also extracted all possible streams to be created between instances of specific manifolds and atomics during operation calls in order to carry the needed data (needed input parameters, possible return values). Guards for receiving of operation calls were also extracted. In the sequence diagrams we will:

- Define specific instances of manifolds and atomics and specify when and by which process these are activated/terminated.
- Put previously created streams in proper states of manifolds and specify the sequence of them in a state. Before adding created streams to states we define the specific names of source and target process instances, since during architecture modeling mapping instance names are not available.
- Specify when and under what conditions or in which state events previously defined by manifolds/atomics are raised.
- Identify which manifolds/atomics receive these events and what are the reactions to a specific event.
- Specify additional actions taking place under specific scenarios (and their sequence) using special messages provided by sequence diagrams such as the writing to ports by a manifold/atomic, the installation of a timer, the assigning of a local variable. etc.

The general steps for mapping sequence diagrams are the following ones:

1. Identify sequence diagrams of each component. These are the sequence diagrams in which the parent lifeline represents an instance of this component.
2. Map one by one the sequence diagrams of each component in the following way:
   - 2.1. Order the messages/actions contained in the specific diagram from top to bottom irrespective of the lifelines that are attached to.
   - 2.2. Group messages to states as follows:
     - 2.2.1. Starting from the first message get all messages in order until next "state transition message" where a "state transition message" is: the next "signal" message sent by a lifeline other than a parent component OR the next "signal" message sent by the lifeline of a parent component to itself OR next "operation call" message received by a parent component from its outside environment OR next timeout event of a timer installed by a parent
     - 2.2.2. Create a state with *label="name of first message"* and put all messages until (excluding) the "state transition message".
     - 2.2.3. Get the next message from the diagram. IF this is NOT a "state transition message" then go to step "a" setting the label of the next state, i.e. name of first message="name of state transition message". *ELSE* get all subsequent messages until you find a message

that is not a “state transition message”. Then go to step “a” setting the label of the next state, i.e. name of first message=“name of state transition message1 & name of state transition message2 & .. & name of last transition message found”.

2.3. Map messages of each state to Manifold constructs using our mapping rules (due to space limitations the mapping rules are not presented here)

3 The Code Generation Tool

Based on our methodology, we have developed a tool that automatically generates the Manifold code implementing the coordination-communication part of software architectures modeled with UML 2.0. Our code generation tool takes as input an XMI document describing the architecture model of a system and outputs the full Manifold code implementing the coordination part of the system. The full route of creating and transforming a software architecture model to Manifold code is shown below:

The creation of the software architecture of the system forms the first step. For the modeling of the software architecture we use the Sparx Enterprise Architect modeling tool (Sparx Systems Website 2006). Using the “export” function of Enterprise Architect we then export the modeled software architecture to an XMI (v.1.1) document.

Since the latest version of XMI (v.2.1) that corresponds to UML 2.0 has recently been released, the few tools that provided support of UML 2.0 after its official release on 2003 have used previous versions of XMI format to export the models and added custom extensions to cover the needs not supported by these versions. Additionally, since XMI has to be general enough to represent not only UML models but every kind of model, there are specific needs of UML tools that may not be supported. As it is stated in (Laird 2001) "the XMI standard itself doesn't support all that is needed, and vendors unfortunately implement it differently". In order to make our code generation tool more independent from specific UML modeling tools we first parse XMI generated by Enterprise Architect and create an intermediate, tool independent, representation of the model. The intermediate representation consists of generic UML 2.0 Java classes that represent the elements of our software architecture model.

For parsing the XMI document and creating the UML 2.0 object model we use Apache Commons Digester (Jakarta Commons Digester Website 2006). Having an intermediate representation of the software architecture enables the support for additional modeling tools in the future with minimum effort. If we wanted to add support for a modeling tool other than Enterprise Architect that has a different implementation of XMI format, then we would only have to add another set of digester rules for parsing the XMI document exported by this tool (or just the rules for parsing the XMI parts that are implemented differently in this tool) and transform it to the common UML2.0 object model.

The next step is the transformation of the UML2.0 object model to the equivalent Manifold object model by applying the mapping rules of our methodology. The

Manifold object instances are finally processed to generate the Manifold code by applying the syntax rules of Manifold.

4 Evaluation-Contribution, Limitations and Future Work

The general approach of the work presented here, as well as the work presented in (Papadopoulos et al. 2006), is the integration of software architectures and coordination models, which enables us to derive the advantages that both of them provide in reducing the costs of the software development process. The modeling of system architectures enables developers to define the more important properties and constraints of the system under development, but also to detect errors early at the design time, thus saving development time. The generated code, which is consistent with the previously modeled architecture, clearly separates the communication from the coordination parts of the system, making the system maintenance much easier.

Furthermore, the integration of software architectures for specification, with coordination models and languages for implementation, has a number of advantages for both software architectures and coordination models as these are elaborately described in (Papadopoulos et al. 2006). In summary: (a) coordination models offer to software architecture an alternative approach to code generation which enjoys the fundamental advantages of coordination models, such as programming language independence (components may be written in different languages even within the same application), and higher degree of component reusability (because of the clear separation of the coordination code from the computational one). (b) Software architectures offer to coordination models a way of modeling and analyzing a system well before its implementation begins. This work enhances the coordination languages with a GUI front end, which can easily be learned and facilitated to generate most of the coordination related code.

In our first work we use ACME, a generic ADL in order to show the generality of our approach. In this way we provided a general route for mapping any particular ADL to coordination code. However, this choice had leaded us to some limitations. Our first methodology cannot generate the code implementing the dynamic configuration of a system. We added a limited support for dynamism by defining the *active_on* property that specifies which event triggers the construction of each possible connection of a system's software architecture.

Based on the results and experience of our first work and exploiting the improvements realized by the latest version of UML towards the support of software architecture descriptions, we have built our second methodology which automates the transition from software architecture design of component-based systems described in UML 2.0 to coordination code. Our second work provides an improved support for the dynamic aspects of the software architecture exploiting the powerful tools of UML for dynamic behavior descriptions as well as the improved support of UML 2.0 for interrelating structure and behavior-centric diagrams. Our second methodology is also supported by an integrated code generation tool that fully automates the production of the complete code implementing the coordination-communication part of software systems modeled with UML 2.0.

Other advantages introduced by our latest work are the following ones:

- The way that the dynamic parts of a software architecture are described by software architects in the latest methodology (i.e. by describing execution scenarios using sequence diagrams) is much closer to their way of thinking than the definition of the active_on property previously used. The methodology makes this process easier by giving guidelines to software architects for identifying all possible execution scenarios and describing them in a hierarchical way.
- The use of a standard, broadly accepted and established modeling language for describing software architectures. Although ADLs have evolved and matured considerably over the last few years, UML is the standard notation language for analysis and design of a system. UML is more familiar to software developers and the one that is supported by many commercial tools. UML 2.0 and XMI standards that we use in our latest work are also two of the main standards proposed and broadly used by the new general software development approach of MDA.
- The two types of diagrams that are used in our methodology can be perfectly interrelated, thanks to the new feature of UML 2.0 for structure and behavior gross integration. The model elements specified in the architecture diagrams such as components, classes, connectors, interfaces are then directly associated with the model elements included in sequence diagrams. This provides us with the ability for automatic consistency checking between elements specified in diagrams, but also makes the development of our tool easier since we did not have to proprietary define the relationship between the model elements of the diagrams that we use.
- By virtue of XMI, the software architecture descriptions can be exchanged and used/edited by many modeling tools. Although we used a specific modeling tool for software architecture design, we have designed our code generation tool in a way that additional modeling tools can be supported in the future.
- Adhering to the main principles of the MDA approach, we tried to keep the software architecture model constructed by our methodology “platform” independent. A "platform" is meaningful only relative to a particular point of view. Since the underlying implementation paradigm that our work is based on, is the coordination paradigm and specifically the IWIM coordination model, we define as “platform” a specific coordination language that implements the IWIM model, such as Manifold. The constraints defined by our methodology on the use of certain UML 2.0 constructs were based on the IWIM principles and not specifically on the Manifold language. In this way the same model can be used to generate code in another language implementing the IWIM coordination model.

The software developer that will use our methodology and the associated code generation tool will face a common, in the field of automatic code generation, problem: the maintenance of the generated code. Although in our latest methodology the coordination code that can be generated is more complete limiting the need for the programmer to manually add missing bits of coordination code, if the software archi-

tecture of the system changes in a subsequent stage (e.g. the system is extended with new functionality and subsequently new components) the code has to be generated again. However, the problem is limited to the atomics files that the tool generates for the coordination-related code and where the programmer manually adds the computational code.

Our future work involves the enhancement of our code generation tool by:

- addressing the problem of code maintenance; we are currently in the process of considering code-block recognition methods used in other code generation tools,
- supporting additional modeling tools apart from Sparx Enterprise Architect,
- adding enhanced mechanisms for consistency checking and validation of the imported software architecture model.

Acknowledgments

The authors of this paper would like to thank their partners in the MUSIC-IST project and acknowledge the partial financial support given to this research by the European Union (6th Framework Programme, contract number 35166).

References

Arbab F., Herman I. and Spilling P.(1993) An Overview of Manifold and its Implementation, *Concurrency: Practice and Experience* 5 (1), pp. 23-70.

Garlan D., Monroe R. T. and Wile D. (2000) ACME: An Architectural Description of Component Based Systems, *Foundations of Component-Based Systems*, Cambridge University Press, pp. 47-68..

Jakarta Commons Digester Website (2006) http://jakarta.apache.org/commons/digester.

Laird C. (2001) XMI and UML combine to drive product development, IBM Whitepapers, available at http://www-128.ibm.com/developerworks/xml/library/x-xmi/.

Medvidovic N. and Taylor R. N. (2000) A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering* 26 (1) 70–93.

OMG MDA Website (2006) http://www.omg.org/mda/.

Papadopoulos G. A., Stavrou A., and Papapetrou O. (2006) An implementation framework for Software Architectures based on the coordination paradigm, *Science of Computer Programming* 60(1): 27-67.

G. A. Papadopoulos and F. Arbab (2001) Configuration and dynamic reconfiguration of components using the coordination paradigm, *Future Generation Computer Systems* 17 (8) 1023-1038.

Sparx Systems Website (2006) http://www.sparxsystems.com.au/.

OMG (2003), Unified Modeling Language: Superstructure version 2.0.