Relatório Técnico

Núcleo de Computação Eletrônica

Enhancing UML Expressivity Towards Automatic Code Generation

Pais, A. P. V.
Oliveira, C. E. T.

NCE - 08/01

Universidade Federal do Rio de Janeiro

Enhancing UML expressivity towards automatic code generation

Pais, A.P.V., Oliveira, C.E.T.
NCE - UFRJ - Universidade Federal do Rio de Janeiro
email: carlo@ufrj.br

Abstract. UML has turned out to be a great tool to exchange ideas among designers from abstraction to detailed design. When it comes to machine interpretation, UML description lacks formalism, coverage and detail to produce a fully fleshed information system. Extensibility and genericity already built-in in the language can be exploited to cater for its deficiencies. UML diagrams can be refined and reengineered to cover unattended areas and missing information necessary for automatic system generation. GUI design, control logic and persistency can be tracked from robustness analysis diagrams down to generation of extended state, sequence, class and object diagrams. These diagrams are enhanced with new stereotypes and tags to enable machine generation of interchangeable UI paradigms, use case controllers and deployment of server entities. This enhanced UML concept is being tested in the development of a real large system using a customized set of scripts in a CASE tool.

1 Introduction

1.1 High level abstraction x implementation

Design level modeling is an abstract specification to guide implementation. Capturing the ideas from analysis level, it introduces refined detailing, targeting a particular implementation. Refinements narrow down development, pinpointing an elected course from a universe of possibilities. A high level of abstraction is applied to design modeling to keep light the burden of developers, and to limit the losses associated to unavoidable necessity of redesign.

Design abstraction presumes hidden information, based upon a preconceived paradigm, supposed to be a commonality of universal knowledge. However, this assumption can detract the stage of acculturation attained by the local developers community. Semantics of the same abstract description conveyed by distinct programmers may not crystallize to the same implementation as intended by the designer.

Insufficient information, both stated on abstract description and instilled in developers, leads to widen the gap between project and implementation. Implementers are expected to fill the bulk of missing information, which may stem from their individual point of view concerning the given description. Moreover, due to terse statements on specification, implementation may lack from requirements abstracted from design logic. As an unfortunate consequence, implementation preferences that might be present at early analysis stages may get lost down the development stream. Subjectiveness still can be handled by appropriate training of involved developers, but can impose insurmountable obstacles to automatical interpretation of design data.

1.2 Extending UML

Taming UML [1] to automatization involves exploiting its extensibility and advancing towards refinement and reinterpretation its specifications. Extensibility is already provided in UML by judicious use of tags and stereotypes. Refinements can be brought forward filling in most available attribute fields and recursing down into finer grain specifications. Reengineering stems from the fact that UML is a tool and not a modeling paradigm. From this, follows that many diagrams can be used in different way than originally proposed. Reengineering of semantics can be applied, reusing a diagram paradigm to specify details missing in existing representations.

UML is defined in extensible way, so that new specifications can be covered until a new release can effectivate them. Stereotypes are used in this proposal to narrow the meaning of otherwise generic elements, helping scanning algorithms to generate appropriate code. Aspects like transactional contexts and access scopes can be specified with customized tags.

Sequence diagrams [2] can exemplify the use of refinement for accurate specification. They picture a coarse idea of object interaction, with hints about branches and loops. Operation in between message issues is completely abstracted from this diagram. Refined diagrams can be introduced to depict happenings overlooked by a particular diagram abstraction. Since messages trigger state changes on the destination object, a state diagram [2] can be associated to each message, detailing operation development, including branches and loops. The same applies to the state diagram, where interaction between objects is weakly represented by

outgoing actions. Sequence diagrams can be associated with transitions, revealing deep object interaction. This recursion can go onwards until sufficient detail is gathered to produce an unambiguous specification.
Semantic tracking to design intention usually transcends the resources offered by cursory interpretation of available UML diagrams. Amending this shortcoming, we can resort to reengineer the semantic representation of existing diagrams to match the desired meaning. Illustration can be given in the refinement of object diagrams [2]. Entity role in use case design [3] entangles the mapping of user intervention on domain objects. This subsumes a correlation between entity properties and user interface. At this point we can expand an object diagram to represent each entity property as standalone instance. The next step is to associate each property to a corresponding visual component representing it in a view-model paradigm. Reinterpreting this diagram we can build a GUI prototype by arranging and nesting visual components instances in a relative positioning layout. On the previous example of state/sequence recursion, refinement can be carried down to statement level specification. A hierarchical document model tree can represent statements in a method specification. A reengineered state diagram can represent this tree by nesting its levels in composite states and sibling nodes as a chain of state transitions.

1.3 Contract, aspects, metaprogramming

Exponential grow of complexity in information system is pushing abstraction to spiraling higher levels. Static representation offered by UML is falling short to the abstraction needs of embracing architectures devised to take specification pain out of the hands of designers. Design patterns [4], aspect programming [5] and contract frameworks [6] are superimposed layers of abstraction escaping further from UML representation realm. As an example, a visitor pattern [4] dynamically encompasses methods for each descendant of the visited root class. Although we can represent a static picture for a predefined set of descendants, the architectural semantics are not captured. Aspect programming is at large, embedded in enterprise application servers. System designers are alleviated from extrinsic constraints to their business logic and UML fits them well. However, server designers are faced with the challenge of representing a system that generates new classes defined by user specifications.
Abstraction extrapolates over to metaprogramming [7] and evolving software in genetic programming [8]. Since abstraction tendency is getting higher, a movement towards finer grain representation appears as traveling backwards on the technology stream. The answer is to retrofit abstraction tools into the process of capturing design intention. This paper describes a set of representations that can be extracted from high-level representation and fed back to designer interaction. Details are inferred from a match of high-level statements against features of a generic architecture. The high level abstraction is a reengineered robustness diagram [3] that is superimposed to an architecture based on a computer assembly metaphor. Subsequent refined diagrams are automatically extracted with simple heuristics scanning the robustness analysis. The whole range of diagrams is capable of expressing sufficient details passive to automatic code generation

2 Description

2.1 Robustness Diagram

The robustness diagram is a result of the robustness analysis [3], introduced by Ivar Jacobson in 1991. This concept involves the analysis of a use case description and the discovery of the initial classes that participate in the use case. These elements are classified in three main categories: user interface elements, entities and controls.
User interface elements are those used to communicate with the user that interacts with the system. Entities are objects that belong to the application domain, and are the result from the domain analysis [2]. These classes represent all the information and concepts manipulated by the application. Control elements establish the communication between the user interface and the entities.
The robustness diagram is a graphical representation of a use case script. The same information contained in an use case script must be obtained from the corresponding robustness diagram. By reading both parts, the user can have a precise understanding on how the system will work. The robustness diagram shows how the classes interact with each other during the execution of each one of the steps described in the use case.
The robustness diagram, like any other diagram, can be seen as an expression of ideas using a language built on top of a set of symbols and formation rules. Its simplicity eases its construction and understanding, but

sometimes it can not express all the meaning contained in an use case. This work proposes a new set of symbols and rules to turn it into a more complete language. Therefore, the creation of new stereotypes and association rules for the robustness diagram makes it more meaningful, increasing its capacity to represent with symbols the ideas expressed in an use case script.
The conception of the robustness diagram is ideal for modeling systems that implement the MVC architecture [9]. The model is responsible for storing all the information about the current state of the application. The view determinates the visual representation of the information contained in the model. The control is responsible for behavior, stating how and when the application must react to user input. Each one of these parts is represented by one of the categories in a robustness diagram: entities, user interface elements and controls. Consequently, there are three symbols that represent these categories: entity, boundary and control [2]. The introduction of new symbols requires a new architecture to bring a context for then.
In other words, these stereotypes externalize the MVC architecture in a simple way. The description contained in a use case can easily overshadow the expressivity provided by the three basic stereotypes. To obtain a model closer to natural language, we need to develop a more detailed architecture. This architecture must reflect the planner's intention of building a flexible, efficient and consistent system. The elements of this architecture are then represented by new stereotypes that can capture with more precision use case details like events, decisions, iterations and exceptions.

2.1.1 Stunt architecture

The Stunt architecture is based on the proxy design pattern [4], where a set of objects impersonates the real entities, their properties and mutual behavior. The element names derive from a computer assembly metaphor, denoting how the parts fit together. This metaphor comprises of a Rack where there are some Slots. Plugged in these Slots are Boards fit with some Chips. These elements in this context have a very specific role in the MVC paradigm, leading to the creation of stereotypes that corresponds to some of them.
RackStunt represents the use case. It is the use case controller and has a state machine associated to it. Each state corresponds to a SlotStunt object. When a state transition occurs the user notices this in the interface because, normally, this transition results in changing the screen. The SlotStunt is the element responsible for showing user screens. BoardStunt objects represent entities, which participate in the use case. Some attributes of an Entity are usually represented in the user interface as read only forms or as editable forms. ChipStunts correspond to these attributes graphic representation, and are aggregated by the corresponding BoardStunt.

2.1.2 Stereotypes

The original Robustness Diagram has three stereotypes: boundary, control and entity, that corresponds respectively to: interface elements, control elements and entities. This work proposes to add the following stereotypes: rack, slot, board, guard and boundary action that can be understood as specialization of the original stereotypes.

Figure 1 - The original MVC stereotypes

Figure 2 - The extended "stunt" stereotypes

Boundary action is a visual element that represents a user action that is able to change the system state. For instance, a confirmation button is a boundary action. When this button is clicked, the form data is checked and a new screen is displayed to the user. Besides, a boundary can aggregate one or more boundary action.
Rack, slot and board come from the Stunt architecture, corresponding respectively to RackStunt, SlotStunt and BoardStunt. They represent specialized control elements with roles determined by Stunt architecture.

Rack represents the use case. Slot is responsible for displaying the corresponding screen when it is the current state. Board is responsible for making the entity available to user interaction, by saving and retrieving the entity data.
Guard is a control element that indicates a decision. It corresponds to the evaluation of a condition, which may be true, or false. So it is associated to two control elements that represents the "true" and "false" paths.
The use of all these stereotypes in the Robustness Diagram showed that the modeling process of some use cases needed two other stereotypes: exit and exception. They ease the understanding of the use case, and offer some details that help the automatic generation of diagrams and codes.
Exit is not a system object; it only indicates the end of the use case, its exit. It helps the designer to view the use case execution flow, showing actions that lead to the end of the use case. Exception represents a exception state, determined by the developer when a business rule is not respected.

2.1.3 Association rules

The creation of new stereotypes for the robustness diagram suggests the creation of new rules, which guarantees the MVC architecture concepts, and, more specifically, the Stunt architecture concepts. This keeps the diagram contents in conformity to Stunt architecture, allowing the automatic generation of the correct code.
There are some rules that have fundamental importance in the construction of the robustness diagram. These rules are briefly described and discussed afterwards.
There is only one Rack on the robustness diagram, once that it represents the use case itself. This Rack must be connected to the slot corresponding to the initial state of the use case. This helps to identify the first use case screen, serving as the starting point for scanning the whole diagram.
The Slot must be attached to a boundary, identifying which screen the slot has the responsibility to show, when it is the current state of the application.
The board must be connected to only one entity. It is only necessary when attributes of this entity are shown in the visual interface.
The boundary action can only invoke elements that fit in the category of control. This follows from the MVC architecture concepts, which determines that the view must send the user events to the application control.

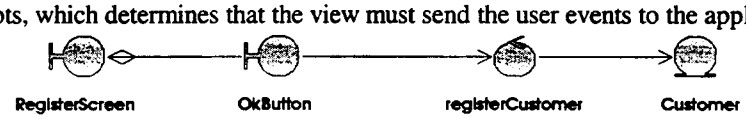

Figure 3 - MVC sample Robustness Diagram

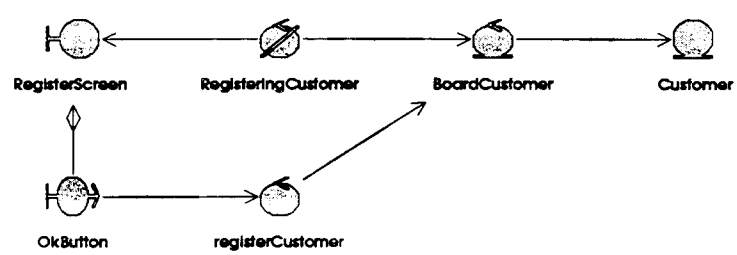

Figure 4 - Corresponding stunt diagram

2.2 Automatic diagram generation

The Robustness Diagram is the bridge between the script, the use case and all other modeling diagrams. From the interpretation of its elements and its associations, it is possible to extract information that will be detailed by the generation of a set of logical diagrams. Since there are well-defined rules to these diagrams, it is settled a relation between the elements of different diagrams, making automatic diagram generation possible, and assuring the consistency between them during the modeling process.
By interpreting Robustness Diagrams, it is possible to automatically generate State Diagrams, Sequence Diagrams and Class Diagrams. Furthermore, it is possible to generate Screen Diagrams, which are extensions introduced by this modeling process.

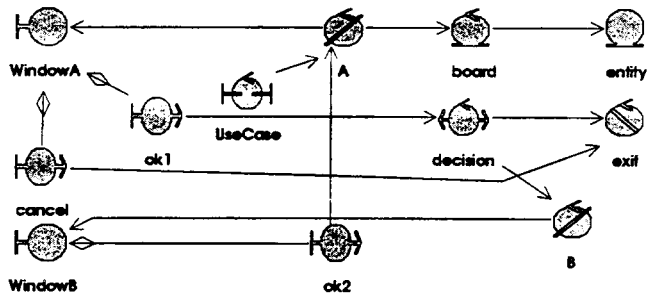

Figure 5 - Sample stunt diagram used for automatic generation

2.3 Screen diagram

The user interface is an important feature in the development of successful Information Systems, although UML barely encourages it. So, to treat this deficiency, the Screen Diagram [2] was added to the modeling process. In fact, the Screen Diagram is the reinterpretation of the Object Diagram. It contains only the visual elements set into user interface, organized as they will be presented to the user. The Screen Diagrams reflects the System visual interface.

The objective of the Screen Diagram is to describe the user interface abstractly, since the System may be implemented in any language. The diagram's elements correspond to XUL [10] elements. XUL is a user interface description language.

The Screen Diagram is automatically generated from the Robustness Diagram's interpretation. The analysis of elements such as boundary, boundary action, and its aggregation, indicates the use case screens and its content such as buttons, menus or any other visual element.

There are other Robustness Diagrams' elements that can be shown by the visual interface. Entities can be presented, usually, as forms. A new record, for instance, can be inserted in the database after the user has filled in a form. Entities have a set of attributes, and a subset of these attributes may be shown by the user interface. These attributes, defined in the diagram, correspond to ChipStunt objects and may be labeled by a set of information, such as: identification of its screen and its XUL element type. An entity representing a employee, for instance, has an attribute called Name. An edit form may show this attribute as an input text field. On the other hand, the screen that confirms the edition may present the same information as a label.

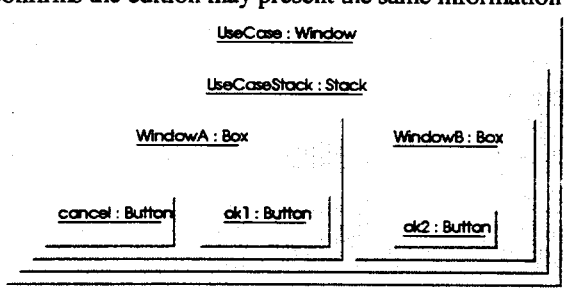

Figure 6 - Screen diagram generated automatically

2.4 Connection diagram

The connection diagram, proposed in this work, contemplates the mapping between the entity attributes and visual elements. This diagram defines a relationship between ChipStunt and an XUL type, marking connection between slots and boards. A slot is represented as a set of all XUL elements that can be presented in the user interface corresponding to the slot. A board is represented as a set of attributes that belong to an entity corresponded to this slot. Each element of a slot is bound to one or more elements of a board.

ChipStunts represent each attribute of an entity. The type of the XUL element related to the slot defines how the attribute will be represented in the user interface and corresponds to a graphic element, representing the

attribute view. The ChipStunt represents the model to this view. It has the responsibility of store the information presented by the corresponding graphic element.
The connection diagram facilitates the process of defining relationships between attributes and visual elements. This diagram can be automatically created from the robustness diagram, inspecting the way that slots and boards are added to the new diagram. From this point on, is the designer responsibility to add elements to the slots and establishing the required connections. Once the connection diagram is done, it can be interpreted, associating the attribute tags to the boards in the robustness diagram.

2.5 Class Diagram

The elements defined in the robustness diagram can be translated into classes or methods. This depends on the designer interpretation. The automatic generation of the class diagram is based in some rules applied to the robustness diagram, which indicate how a element ranks as class or method. Once the elements classification is done, the rules define the class to which each method belongs.
Elements in the robustness diagram can be divided in three main classes: model, view and control. Model and view elements are likely to be defined as classes. Control elements, however, are not so easy to classify.
New stereotypes allow a better understanding of the control elements behavior, following the Stunt architecture. So racks, slots and board elements can be seen as specializations of the base classes RackStunt, SlotStunt and BoardStunt. The remaining control elements are classified as methods.
Some simple rules were established to define the association of methods to classes. Elements identified as methods corresponding to boards are translated as BoardStunts methods. Elements related to an entity are defined as BoardStunt methods corresponding to the board bound to the entity. If there is no board related to the entity, the methods will belong to the class that represents this entity. The remaining elements will turn into RackStunt methods corresponding to the rack, which is unique in the robustness diagram.
From the classes that represent use case entities, it is possible to automatically generate the deployment to the persistence for EJB containers [11]. This deployment is a XML [12] file that configures the object persistence. In order to do that, it is necessary to detail the entity attributes, as for instance, the write and read access.
The entity attribute can be better specified in the robustness diagram by adding tags. These tags can be read-only, write-only, or finder. The finder label defines if the attribute value can be used as a key to query one or more elements in a collection. So, for example, if attribute name of a student entity is defined as a finder, it can result in the creation of a findByName method that query students based in their names. The other labels specify how the corresponding attribute can be accessed, which can be read only, write only or read and write.
The automatically generation of the persistence files involves only the recognition on the entities represented in the different use cases. The following step is to evaluate the associations between entities and attribute tags. The information collected in the diagrams is enough to completely generate the persistence deployment for the system.

2.6 State Diagram

The state diagram generation is possible through the interpretation of the relationship between some of the elements in the robustness diagram. The identification of these states is a good starting point to this interpretation. Once it is done, it is necessary to identify, in the robustness diagram, how the transition between states happen, as well as, the events that cause them. Finally, a preliminary state diagram that includes the information represented in the robustness diagram is created. However, this diagram still has to be refined by the designer.
The state of the application can be related to an element in the robustness diagram that will be translated in the state diagram. From the user point of view, the better candidate to a state is an element that represents a screen in the user interface. Each screen change represents a transition of states in the system. So, the current screen represents the current state of the application. The screen is represented by a boundary. This element is bound to a single slot that controls the boundary visibility. So a slot is translated to a state.
A boundary action of a robustness diagram is tightly related to an event in the state diagram. It represents a visual element that can cause a change in the application state. This element represents an event fired by the user. This justifies the tight relationship between a boundary action and an event. Both can be interpreted as elements able to cause a state change in the system. So, a boundary action is translated to an event.
After the states and events identification, remains identification of transitions. Transitions can be obtained by interpreting robustness diagrams as an oriented graph. The elements are the nodes and the associations are the

arcs. A path between two slots containing a connection with a boundary action can identify a transition. Navigability stated in associations determines the transition orientation. Guard stereotypes encountered alongside the path are accounted for guard conditions.

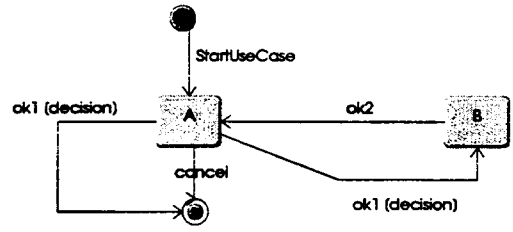

Figure 7 - State diagram generated automatically

The transitions from the automatic generation are sometimes excessive, since some transitions may not exist. In this case, the designer determines which transitions are false and take them away from the diagram that was automatically generated. The other transitions are incomplete, because the actions associated to them are not identified.

The heuristic used to recognize the transitions, from the Robustness Diagram, is very simple. This precludes the identification of actions. Any control element between two slots, identified as a transition, may correspond to an action, although there is no stereotype that indicates a control as an action.

As said above, the State Diagram automatic generation results in a incomplete state diagram, but this diagram is consistent to the Robustness Diagram. The diagram is incomplete because actions corresponding to transitions are not identified and some transitions may not be correctly identified. Nevertheless it keeps consistency with the Robustness Diagram. After the automatic generation, the designer can improve the State Diagram. During this process, the designer may add new states, transitions, etc. The creation of new states may indicate that the Robustness Diagram is not complete and need to be reevaluated, resulting in an improved use case.

2.7 Sequence Diagram

The sequence diagram, as well as the robustness diagram, shows how the classes interact during the execution of the use case. This diagram represents the stream of messages between the classes found in the use case. Its automatic generation is based on the robustness diagram semantics and on the stunt architecture concepts.

The first stage for the automatic generation of the sequence diagram is the identification of the involved classes in the use case. The elements of the robustness diagram are classified in class or methods, using the same rules for the generation of the class diagram. The identified classes are added to the sequence diagram.

The messages contained in the sequence diagram correspond to the methods identified in the robustness diagram. Each one of these messages is constituted by an origin and a destination. The origin is the class that invokes the message, while the class that contains the corresponding method is the destination. The destination of each message is identified directly from the robustness diagram analysis in agreement of what was described in class diagram. On the other hand, the origin must be determined taking into account the underlying architecture.

The stunt architecture focuses on the implementation of the use case as a finite state machine, represented by the RackStunt. The RackStunt receives stimulations from the user interface, which are transformed in events of its state machine, causing transitions. The SlotStunt has the responsibility of receiving the event, making the necessary verifications, executing the corresponding actions and indicating which is the resulting state for this transition. Since the methods identified in the robustness diagram are related to the verifications and occurred actions in the states machine transitions, the origin of these messages is the SlotStunt, corresponding to the source state of the transition. Moreover, some inserted messages in the diagram correspond to methods defined in the classes of the stunt architecture, which were added envisioning the production of a cleaner diagram.

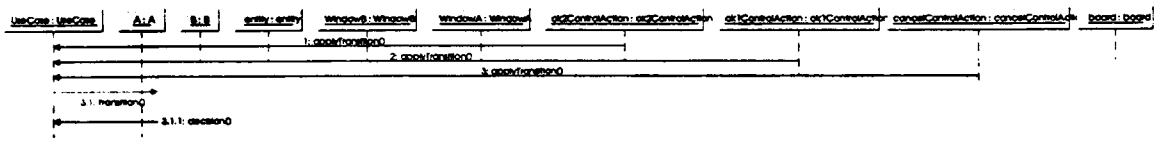

Figure 8 - Sequence diagram generated automatically

3 Assessment

3.1 Extension of the robustness diagram semantics

The extension of the robustness diagram semantics increased its capacity to capture the ideas contained in the use case. This was the starting point to establish a correlation between its elements and the elements of other diagrams, envisioning the automatic generation of logical diagrams and source code. The automatic generation maintains the consistency among the several diagrams, during the whole modeling.

The proposal of this work was tested in the modeling and implementation of some modules of the SIRA system, that is an integrated system for academic registration, implemented in Java [13]. As a consequence, the automatic generation was targeted to Java source code. The case tool used in the system modeling was ObjectDomain [14]. This tool is written in Java, and it allows the access to the internal metamodel through external modules. This facilitated the automatic generation implementation through the use of pluggable modules.

3.2 Predefined architecture

The robustness diagram, although reflecting the use case semantics, does not conform to system modeling based on architectures deviating from the MVC, for example, the view-model architecture. Some systems, due to its simple nature, suggest a model of simpler implementation, defrauding the concepts of the MVC architecture. Frequently, the planners opt for view-model architecture, which promotes a direct connection between view and model, where the view assumes the functionality of the control. In this case, the system implementation can be simplified, and executed quickly.

Web applications are a good example of view-model architecture, where persisting the application current state is not necessary during the user's session. The user navigates through random pages, filling data forms stored in the database. In this case, the current state of the application doesn't matter, and the browser session is used to identify the user during its surfing through application screens.

In order to make robustness diagrams suit these architectures, a new group of rules can be created, or the existent rules can be relaxed under certain circumstances. Relaxing the rules would allow a view element to invoke another view element directly, without having to pass through the verification of a control element.

The automatic generation of these systems demands the design of an architecture based on the view-model paradigm. This architecture could be a simplification of the stunt architecture. The control elements tend to disappear, once the control of the application is delegated to the view. Besides, the use case doesn't need a state machine, eliminating SlotStunt from the stunt architecture and the state diagram from the model. These simplifications reduce the modeling complexity, as well as they facilitate the automatic generation of diagrams and code.

3.3 Source code generation

The automatic generation process started in the robustness diagram analysis maintains the consistency among the several diagrams, up to the moment when the diagrams contents are translated into source code. The automatic generation implemented in this work brings benefits to the planner during the modeling task, accelerating the construction of the diagrams. In spite of that, the planner sometimes has to check if the diagram is correct, and, most of the time, he has to add details to the diagram. The objective of this work is to reduce the effort spent in diagram construction, so that the planner just concentrates on the elaboration of the use case. Besides, the greater is the diagrams expressivity, more complete it will be the code generation, accelerating the modeled system implementation.

The result obtained by the implementation of automatic generation process constitutes a great progress in relation to this work proposal. In spite of that, there are still many problems that need to be solved to achieve better efficiency in automatic generation. Some stages of the automatic generation are discussed forward, relating advantages, flaws and possible solutions.

The state machine automatic generation works well in maintaining the consistency between the robustness diagram and the state diagram. But it contains some flaws as the creation of false transitions and the absence of the actions. This could be solved by the addition of elements to the robustness diagram.

The creation of false transitions resides in the simplicity of the heuristic used in the identification of the transitions starting from the elements of the robustness diagram. The problem resides in the fact that certain paths between two slots, which are recognized as transitions, don't correspond to any piece of the use case script. Since the elements of the robustness diagram correspond to ideas described in the script, the construction of the robustness diagram can follow the text of the script, numbering each one of the transitions created. That numeration would facilitate the recognition of the transitions, and also improve the understanding of the interaction among the diagram elements.

The difficulty in recognizing the corresponding actions to transitions can be overcome with the addition of a stereotype that identifies the control element as a resulting action of a state transition.

The generation of a more complete state diagram, that needs a smaller number of modifications by the planner, it is possible through the addition of the elements described above. This allows the planner to concentrate on verifying if is necessary to add new elements to the generated diagram, as, for example, the addition of a new state, that doesn't have a corresponding slot in the robustness diagram, indicating the need of a revision in the use case.

The state diagram contains the state machine description of the use case controller. The source code generation is the result of the state machine elements interpretation. In a general way, detailed diagrams result in complete and functional code. The source code generated from the state diagrams of the SIRA modeling, with minor adjustments, proved to be functional. Screen sequences were controlled in conformance to the use case requirements.

The necessary fittings to the generated source code, most of the time, are related to the business rules implementation. In any stage of the modeling it is possible to specify in details the business rule, in such way that its implementation is generated automatically. This could be possible, increasing the details in the diagrams.

The sequence diagram holds a larger amount of details. It is closer to the system implementation, portraying the message flow among the objects. Starting from its interpretation, it is possible to determine part of the method implementation of the objects, in respect of invocation of another methods. But, this diagram doesn't have enough expressivity to represent loops or condition tests. To supply that need, it would be necessary to add new elements to the diagram, that would turn it more detailed and they would facilitate the generation of more complete code. It is important to observe that the diagram detailing should maintain its construction simplicity and understanding, without bringing larger complexity the modeling task.

The addition of new elements to the sequence diagram can render it too complex. On the other hand, the detailing of that diagram could be made in auxiliary diagrams, which would have the objective of detailing the execution of the methods corresponding to the messages. This way, the planner would associate a diagram to the message, which would consist of the meticulous specification of the message execution.

The execution flow of a method can be expressed in the form of a deterministic automaton, as well as the state machine represented in the state diagram. Following the same philosophy of the screen diagram, the auxiliary diagram can already be obtained starting from the reinterpretation of an existing diagram, which in this case could be the state diagram. To achieve this it would be necessary to establish a relationship among language constructions, as loops and conditional tests, and the state diagram elements. The combination of the sequence diagram with the new interpretation of the state diagram would result in a diagram rich in details, with the potential of being the base for the generation of complete source code.

The screen diagram contains the necessary information for the creation of the application graphic interface. The generation of a XUL document starting from the screen diagram contents is a very simple task, since its structure is similar the XUL structure. The abstract description contained in the XUL document entitles the generation of several representations. The SIRA system supports rendering in HTML and Swing [15], and it implements modules that interpret XUL and render the two mentioned interfaces. This allows similar systems to use any user interface without changing its modeling or control code.

4 Conclusion

The tradeoff between abstraction and objectivity has always been balanced towards the cost effectiveness of subjective interpretation of design level specifications. The main consequence is an augmented distance between represented design and implementation. Skillful programmers, fine-tuned with designers, are required to capture the implementation concept from summary design representations. Conversely, fine grain descriptions can alleviate the effort of following the designer's intention and the risk of misinterpretation.

Conciliating the lightweight burden of high-level abstraction with the objectivity of detailed description, we have proposed an automated generation of multiple diagrams, stemmed from a single representation. Reengineering and refinement of UML diagrams are proposed to cover the whole range necessary to produce a fully fleshed information system. These diagrams cover from user interface to persistency deployment of entities, including control logic. Once generated they can be fed back to the designer, both to access the accuracy of the original design and to allow retouches necessary to match undiscovered or mistaken assumptions.

The system was implemented adding scripts to a professional CASE tool and tested in the conversion of a legacy system. The method was taught to undergraduates, and good results were achieved in few weeks of training, design and implementation.

Enhancing abstraction towards dynamic modeling, capable of encompass design patterns, aspects and evolutive computing is still a challenge. These very same techniques applied under the hood of CASE tools can bring better solutions to the current stage of automation and extend it to cover their own needs.

5 References

1. Fowler M., Scott K.: UML Distilled - A Brief Guide to the Standard Object Modeling Language
2. Page-Jones, M.: Fundamentals of Object-Oriented Design in UML (2000) - Dorset House Publishing
3. Rosemberg, D.; Scott, K.; Use Case Driven Object Modeling With UML: A Practical Approach; Addison-Wesley; 1999.
4. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.; "Design Patterns – Elements of Reusable Object-Orient Software"; Addison-Wesley; 1998.
5. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Proceedings of the European Conference on Object-Oriented Programming, number 1241 in Lecture Notes in Computer Science, pages 220{242. Springer-Verlag, June 1997.
6. iContract: Design by Contract in Java - http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools.html
7. Metaprogramming and Free Availability of Sources - http://fare.tunes.org/articles/ll99/mpfas.html
8. Genetic Programming III: Darwinian Invention and Problem Solving, John R. Koza, Forrest H. Bennett III, Forrest H. Bennett, Martin Keane, David Andre (Morgan Kaufmann Publishers, March 15, 1999)
9. "Developer's Guide - Borland - Jbuilder 2", Borland, 1998.
10. XUL Programmer's Reference Manual - http://www.mozilla.org/xpfe/xulref/XUL_Reference.HTML
11. Monson-Haefel, R., "Enterprise JavaBeans, 2nd Edition", O'Reilly & Associates, 2000
12. McLaughlin, B.; "*Java* and XML", O'Reilly, 2000.
13. Eckel, B.; Thinking in *Java*; Prentice Hall PTR; 1998
14. http://www.objectdomain.com
15. Robinson, M.; Vorobiev, P.; "*Swing*"; Manning Publications Co.; 1999.