Transformations Rules for UML-RT

Rodrigo Ramos

Centro de Informática, Universidade Federal de Pernambuco, P.O.Box 7851
CEP 50740-540, Recife-PE, Brazil
{rtr}@cin.ufpe.br

Abstract. With model-based development being on the verge of becoming an industrial standard, the need for a systematic development based on security transformations necessary. Transformation that leave in account changes in behavioural and structural diagrams. In this paper, we presents a set of UML-RT transformation laws that aid the model based evolution, preserving the behaviour and some safety properties among the model. Permit justify transformation of initial abstract model in analysis to concrete design models, closer of implementation.

1 Introduction

In model driven developments [1,2], consolidated under the name of Model Driven software Engineering (MDE) [3], the central artefact in the development are models, rather than programs coded in a programming language. Different of general idea that the usefulness of models are only for documentation and to capture some of the interesting aspects of the software to be built, the main objective in MDE is that software development process is driven by the activity of modeling the software system, combining a architecture of the models roles (Model Driven Arquiteture - MDA [4]) with the others process activities, and replace the code by model in main process built element.

The OMGs Model Driven Architeture (MDA) [5] defines an approach for software development that separates the specification of system functionality from the specification of how it be implementate on a particular technology platform. The first specification is structured as a platform-independent model (PIM), and in this architecture it can be refined into one or more concrete and platform-specific models (PSMs).

Following this approach, the PIM will potentially have good abstraction and reusability, away from any technical detail, making possible this details be discovered along the design. However in the life cycle of this design is possible apply several kinds of mappings (refactoring and refinement, projection based on plataform, plataform dependent transformations and mining of existing implementations) over theses models, PIM and PSM, most of the paper focuses on the PIM to PIM mapping.

In this framework, transformations on models are important artefact as the models itself. As in general software engineering, refactorings [6,7,8] and refinements [9,10] still help to overcome the challenges of evolution software, defining

software transformations that restructure a software while preserving behaviour and others properties. While refactorings are used to improve specific quality in the model, like applying design patterns in that model, refinement firm basis to concretion of this model. Some proposals for refactoring on models [4] show that operational interpretation on Unified Modelling Language (UML) [11] models is available, despite they general use on code-levels. In MDA, is possible see a clear similarity with concepts of the refinement theory [12,13,10] e com others models approaches that evolving refinement, as the Integrated Computer Aided Software Engineering (I-CASE) [14]. Among this, the validation of the preserved properties in theses transformations still a enormous challenge.

The OMGs [15] Model Driven Architeture (MDA) [5] defines an approach for software development that separates the specification of system functionality from the specification of the implementation of that functionality on a particular technology platform. The first specification is structured as a platform-independent model (PIM), and in this architecture it can be refined into one or more concrete and platform-specific models (PSMs).

Following this approach, the PIM will potentially have good abstraction and reusability, away from any technical detail, making possible other details to be discovered along the design. In the life cycle of this design is possible apply several kinds mappings over theses models [5]: PIM to PIM (refactoring and refinement of designs), PIM to PSM (projection based on plataform characteristics), PSM to PSM(transformations dependent of the plataform concepts) and PSM to PIM (mining of existing implementations for useful PI abstractions). However, most of the paper focuses on the PIM to PIM mapping.

Normally in this development scenario of MDA, models be expressed in the UML, using its profile mechanism to specialize and extend the language for different contexts. The Meta Object Facility (MOF) [16] is used to define the foundation of modelling language syntax and semantics of UML and its profiles, not only in a formal way, but also in way that provides the basis for the mechanism (tools) thought which models are analyzed and manipulated. The integration of several OMG standards in MDA are defined in [5], and its a clear goal of OMG.

In general, the abstractness of the PIM can be expressed using only UML, but others of its extensions could be necessary to express specific aspects independent of platarform. In particular, emphasize the UML profile for Scheduling, Performace and Time [17], commonly know as UML-RT [18,19,17]. This UML extension has active object concepts (capsules and process) and temporal aspects that may need to describe some PIMs domains.

In MDE, transformations are increasingly seen as vital assets that must be managed with sound software engineering principles [20], inclusive methods to the validation. They development are composed of many features, as transformations rules, rule application scoping, rule scheduling, rule organization, rule application strategy, and others [21]. This feature model makes the different possible design choices for a model transformation approach, where transfoma-

tion rules are main of theses feature and one of that are sensible to the model language.

In this context, we propose a set algebraic transformations for UML-RT, that, despite its name, focus on modelling concurrent and distributed aspects. The laws permit justify transformation of initial abstract model in analysis to concrete design models, closer of implementation. they are present as a hierarchic, where basic laws are used to derive more elaborated transformation rules.

A transformation rule consists of a left-hand side (LHS), to accesses the source model, and a right-hand side (RHS), to represent the target model. They are represented using variables (model elements), patterns (model fragments with zero or more variables) and logic computations or constraints. Logic can take a declarative or a imperative form. Examples of declarative form include OCL-queries to retrieve elements from the source model and the implicit creation of target elements through constraints. Imperative logic has often the form of programming language code to manipulate models directly [21].

We decided use this declarative paradigm [20] (know as relational approach group in work of Krzysztof [21]) due to the simpler semantic model required to understand the transformation rules, where transformations are defined by composition of rules described using pre- and post-conditions. Preconditions define patterns that are to be matched in the source model, and are used to identify interesting elements; post-conditions define the state of the destination model once the rule has been applied. This approach covers both OCL specifications and graph rewrite systems. It is both expressive and precise, as the transformation programmer can refine the conditions as much as necessary using a constraint language, and is technology independent. Others major categories can be viewed in the works [21,22].

We use UML-RT because it incorporates a modeling method with clear guideline how diagrams should be applied which UML currently does not, and because it presents some features closer in consistently models, as seamless and wide spectrum [23].

With the objective of show the viability of theses transformation, we realize a initial work [24], that presents a set of laws of large grain applied in desing, reflecting the importance of transformations in practical task of development. This transformations are based on well defined semantics as founded on formal methods, with the intuit of show a semi-formal development where the developer realize his task don't whink in formalisms, but based in them. The formal language used is OhCircus, a object-oriented version of Circus [25], that extends CSP [26] with concepts of Unifying Theories of Programming [27]. Despite the contribution of this work, little laws was exhibit; in pratice is necessary the creation of a complete set of laws which could be easily understated and proved (simple and basic). This set need explain the most of action in the development with UML-RT.

In next section, we presents a overview on UML-RT. In section 3 we present some of us transformations laws. In section 4 we present a study case with the

application of us laws in practical task of a process. And, finally, we presents us conclusion and related works in sectio 5

2 UML-RT

UML-RT uses the basic UML mechanisms of stereotypes and tagged values for defining three new constructs: *capsule*, *protocol* and *connector*. Capsules describe possibly complex active classes that may interact with their environment through messages (input/output signals). To each capsule is associated a unique behavior, given by a state machine. In addition, a capsule can also be defined hierarchically, in terms of compound capsules, each of which with a state machine and possibly a hierarchy of further compound capsules.

Protocols define offering a set of input/output signals that a capsule may have, like services of an interface. And port are protocol instances, declared in capsule, that defining the only possible way a capsule can interact with its environment. Ports can regulate the flow of information (the protocol might have its own state machine to describe this behavior) in communication of capsules. Ports can be public or protected. Public ports allow communication with the environment, whereas protected ports are used for communication with (and among) component capsules.

In general, a protocol may involve several participants (with several roles). Often, however, most applications are confined to binary protocols (involving only two capsules). For a binary protocol, only one role needs to be specified (the *base* role). The other complementary role, named *conjugate* and indicated with the suffix ~ in the base role, can be inferred by inverting inputs with outputs, and vice versa. A port which plays the conjugate role is represented by a white-filled (in opposition to a black-filled) square.

Connectors are used to interconnect two or more ports of capsules and thus describe their communication relationships. A connector is associated with a protocol and acts as a physical communication channel between ports which play complementary roles.

A architecture of a model in UML-RT are composed typically of three diagrams: class diagram, statechart and structure diagram. A UML-RT class diagram includes capsules, classes and protocols of a system, and their relationships. As a convention, we describe invariants, pre- and pos-conditions as *notes* in the diagram, as notes. The main difference between class and capsule is that the first have a bottom compartment is used to declare ports. As already mentioned, the actual behaviour of a capsule is given by a statechart, formed by transitions and states. In general, a transition has the form `e[g]/a`, where `e` is an input signal (or a set of input signals), `g` is a guard and `a` is an action. The transition is triggered by the input signals and a true guard. As a result, the corresponding action is executed. A structure diagram, a collaboration diagram of capsules instances, is used to show the interaction of capsules through connections between their ports. Examples of these three diagrams could viewed in Figure 1.

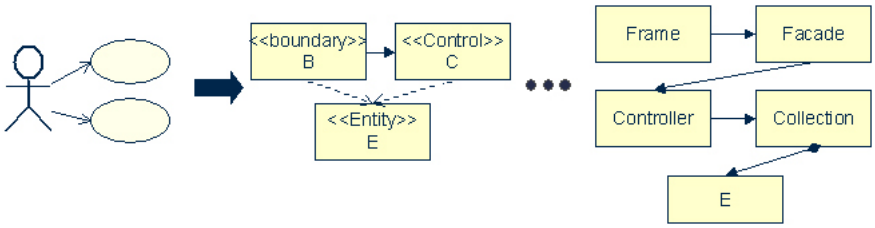


Fig. 1. Diagrams examples in UML-RT: class Diagram(left), statechart(middle) and structure diagram(right)

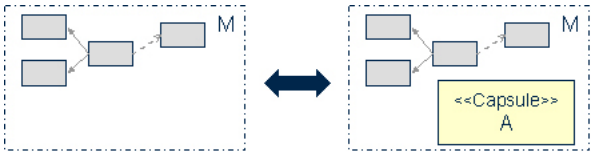
3 Transformations Laws

The objective of create basic laws is contribute to systematization of transformations of a abstract analysis model to a concrete design model, using consistent steps. This steps consider that the notion of compositionality with models in MDA are seemed with in the object oriented achitetures [28]. Like in component-based approachs [29] is possible keep the effects of changes local in the software development. Several problemas of consistency occurs during these local transformations on UML based models (including UML-RT), that in most cases are because the semantic of object-oriented diagrams are still not defined [30]. Exist notions of the refinement theory [12,13,10] are used to support this methodological development.

Thus, our focus is on laws which express on fine grained, isolated refinement steps like removing or adding a single transition in a Statechart or others elements in a class diagram [4,31]. These small set of small grained transformations are simpler to undestand and prove, and they composition are more easy to create well formed design-level transformations.

The four initial transformation rule establishes that the introduction of new elements, with a fresh name, does not alter the behaviour of the model. It also indicates we can always remove a this elements if they are not used by the model. Since in UML we can not have two elements (capsule, class, protocol or interface, attributes) with the same name in a same context (package, capsule or capsule) then we have a proviso stating that name of the fresh element does not appear.

Law 1. *Introduce Capsule*



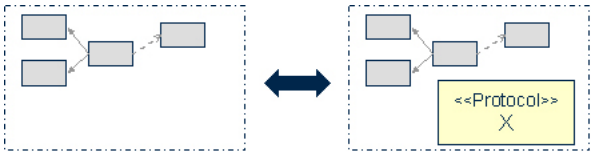
provided

(→) *The model M does not declare any element in same package, whose A will be allocated, named with A.*
(←) *Any capsule in M has a relation with the capsule A in any diagram.*

We write (→) before the proviso to indicate that this proviso is only required for applications of this law from left to right. Similarly, we use (←) to indicate that it is only for applying the law from right to left, and we use (⟷) to indicate that the proviso is necessary in both directions.

When we say *any capsule in M has a relation with the capsule A in any diagram*, means any elements depends, has associated, extends or has a connection with A. Relations where another capsule is a compound of A, are considered too.

Law 2. *Introduce Protocol*



provided

(→) *The model M does not declare any element in same package, whose X will be allocated, named with X.*
(←) *There isn't any port with type of the protocol X.*
(←) *Any protocol extends of X.*

Similar to the Law 1, to remove a Protocol X is necessary that any element use that. In this case the uniques relation with protocol is a association, making a port with this type, and the generalization.

In this two laws the unique diagram modified is the class diagram, because is a proviso that these elements (capsule and protocol) is not used in the model, and so not appear in any other diagram. The next law establishes that we can add or remove a port in a capsule a part the capsule statechart. Over again, we need a proviso stating the name of the fresh element does not appear.

Law 3. *Introduce Port*

provided

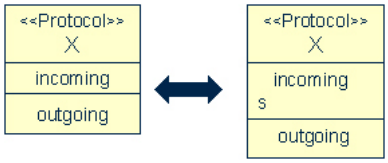
(→) *There isn't any other port in capsule A with the same name of A.*

(←) *Any signal of port is used in a transition or state action of the capsule statechart.*
(←) *Any connection is connect to this port in the structure diagram.*

Port, despite in a different compartment, can viewed as capsules attributes. And for this reason, we can not have two port with the same name in a capsule, we have a proviso stating that the name of the fresh port does not appear. To remove the port is needed a proviso indicating that the is not used in the statechart (*end port* type) and no connect with another external or compounded capsule (*relay port* type). In a general form, this law is very with Fowler refactoring *Add Field* [6].

The next law establishes when is permitted add and remove signals in a protocol. This signal are normally used to trigger event transitions or in actions of states (entry or exit) and transition to send messages to another capsule.

Law 4. *Introduce Signal*



provided

(→) *There isn't any signal (in or out) in protocol X with the same name of s.*
(←) *Any statechart (in the protocol X or a capsule with a port of this type) use the signal s as transition event or in an action.*

The next law establishes when is permitted add and remove a connection. Since we can not have two connection in a port with the same name in UML-RT, we proviso that new connection has a fresh name.

Law 5. *Introduce Connection*

provided

(→) *There is not any connection between ports r and s.*
(←) *The port r and s are not used by their capsules.*

Despite in UML-RT is permitted more then one connection between two port, the behaviour of this is not much understandable, because when a signal becoming only one transition is fired and so the duplicity of signals with more than one connection is not necessary. We think the more security proviso to remove a connection is its extremes port (r and s) are not use, in the statechart of those capsules (*end port* type) or connected to another port of a compound capsule (*relay port* type).

We think that sometimes a port can be used to broadcast messages to many capsule, and in this case may be possible remove a redundant capsule in this comunication, through connection between them. But is very difficult understand is global behaviour behind the communication of several capsules with differents statecharts, for example if are the are index port or not and so on. realizadas

As the law 5, the changes realized by the next law is seen only the structure diagram. This law establishes that a port arity (cardinality) can be always increase, but decrease only its arity is not used at all.

Law 6. *Change Port Arity*

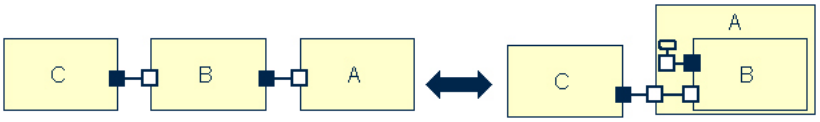
provided

(←) Exist less connections to the port p than the pretended arity.

In this law we only can decreases the arity of a port if has less connection to it tham the final arity value of the port p(in the left model).

Sometimes is interesting move a capsule to inside another, to facility the communication with anothers compound capsules or to manage its execute flow better.

Rule 7. *Move a capsule to a compound capsule*



In this rule, the capsule B is transformed to a compound capsule of A, preserving all the communication of B with others capsules. This rule only affect the structure diagram where B and A stay.

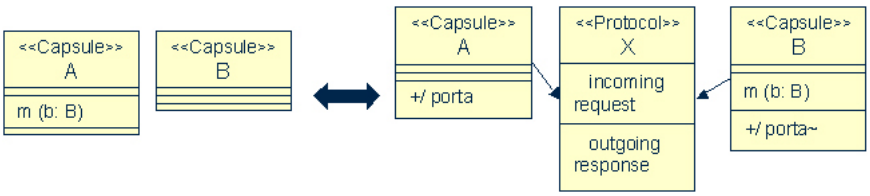
If B has a port that connect with a end port of A(directly linked with the statechart), when B is compounded with A, the port of A may be protected only

if is no more used by others capsules. If the port of A is a relay port type, the B's port will connect directly with anothers ports of the compound capsules, and the port of A is removed (reverse use of the Law 3).

Similar, when we move out a compound capsule from A, protected end port of A will be public and others compound capsule will need of the creation of a relay port to communicate with B, if necessary.

The next rule, move method, is founded in the work of Martin Fowler [6] using between class. The difference here is how to make it with capsules. When a capsule call its methods, the mechanism to call them work equals to in a class. But between capsula, is impossible call a method one of another.

Rule 8. *Move method*



provided

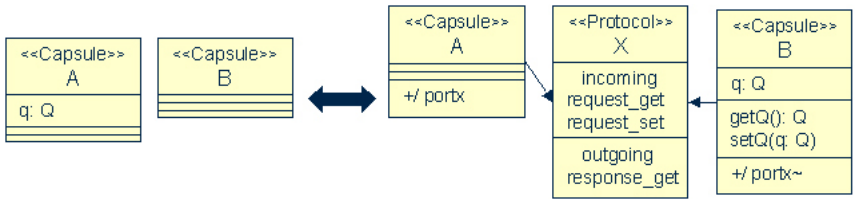
(⟷) There isn't any method with the same name m in the destination capsule.

As we said, the difficult of this rule occur between capsules. When the move destination is a class, the rule is similar to the founded in the work of Fowler [6]. When use two capsules, the new communication between them occur with the change of messages. To make it is necessary to create a new protocol 2 and one port in each capsule 3, where the protocol needs of signals that represent the request and response of the method m computation. In the Bs statechart is needed the additional of a *internal transition* in its outermost state, to it perform the received event without changing its state. In A's statechart needed change the call of method m to a send of request to B and put the rest of the action of this state in another one fired by response, that waits the computation of m in B and will have the out transition that the first state haded.

The next law establishes the move of attributes. This law is founded too on the work of Fowler, but using only classes. Now we show how make it with capsules, using the changing of communication approach showed the Law 8.

Rule 9. *Move attribute*
provided

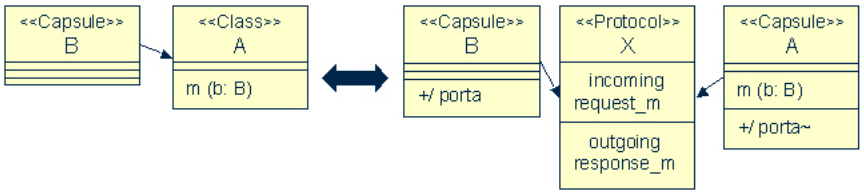
(⟷) There isn't any attribute with the same name q in the destination capsule.



Similar to the Law Move Field of Fowler, us fist step is encapsulate the attribute q with methods setQ and getQ, to update or query the value of q. After that this rule is similar to the rule 8, where the methods are move with the attribute to B.

Motivated by a refinement method with the initial model in the early analysis level. we need a rule that replace a well establish concept of class in the new concept of capsule.

Rule 10. *Replace class with capsule*



provided

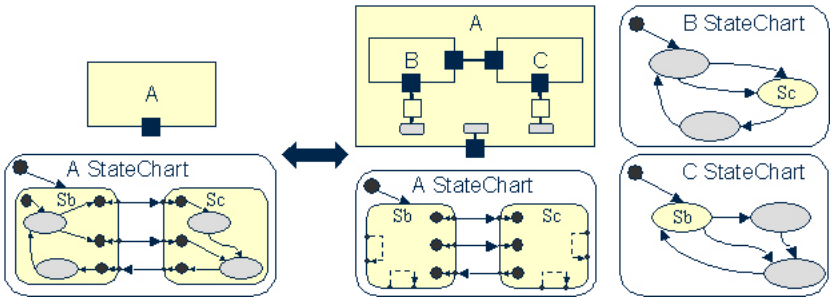
(←) Any other class in the model has a association or dependency to A.

If the class A has a capsule B that uses it, the capsule B will change its statechart to use the methods in A (after its replaced by capsule) in the same way of the approach to request for methods in the Law 8. To do this, the A, need a port of a Protocol com signal to request and response for its methods. The initial statechart sugest to A will be a unique state with transitions with events to each exist request in its protocol, and actions that computet each method and response their return. The old action in constructor will moved to the action in the initila transition.

You can imaginate this rule as a sequence of the Inline Class(Fowler refactoring) A in B , introduce a new capsule A 1 and move attributes and methods of B to A (Laws 8 and 9).

The next law establishes that always we can decompose a sequential composition of states (or-states) in more capsules.

Rule 11. *Or-State Decomposition*



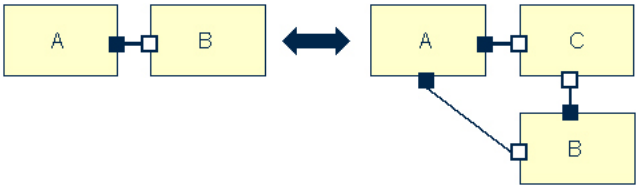
In this law, a capsule `A` has two state `Sb` and `Sc` which we want move out to another two capsule `B` and `C`. To do this we must preserve the behaviour of a Or-State statechart. When the capsule `B` stay in a state move out from `A` (differente of `Sc`), the capsule `C` must stay in `Sb`, a state that represent that B is active and C idle. The reverse is also true for `B` and `Sc`.

So, the statechart of new capsules `B` and `C` will be similar to the statechart of substates of `Sb` and `Sc`. `A` needed of new ports (Law 3) to delegate all arrive messages to `B` and `C`. Events between the states of `B` which fire transition from or to `Sc` will be treated by by internal transitions Sb in A'state. If B use methods in C, we can use the approach explained in Law 8. In a similar way, the capsule `A` manage the signals used in `C`

This rule can be applied to any Or-State statechart. If it don't have hierarchic states (as `Sb` and `Sc`) in `A`, a trivial law that create a superstate can be applied with no proviso and with behavioural preservation of the statechart. For reason of space, this will not showed here.

Motivaded by the Fowler refactorings hiding delegation and remove middle man, we formulate the next law. It exibity haw create a capsule which intermediate the send and receive of signal of a capsule.

Rule 12. *Introduce Middle Capsule*



provided

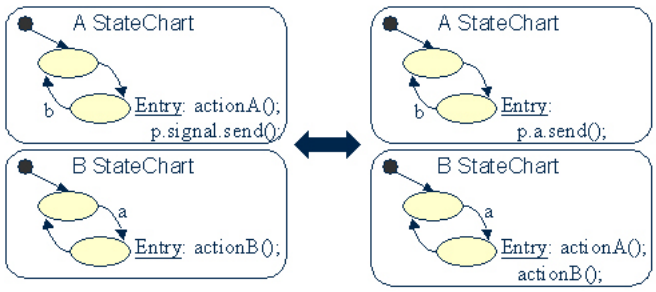
(→) *All states in* `A` *statechart that has in transitions with these signals must appear in* `C`.

(→) *All traces of* `C` *must exist in* `A`*, ignoring the unselect signals.*

In this law, as select set of signals used between two capsules `A` and `B` are intermediated by a third capsule `C`. So, when a one of these signals is send by `A`'statechart, it firt became in `C` and therefore in `B`. The same occur to signals send by `C` to `A` to select signals. The statechart of `C` have the same topology of `A`, considering only the set of select signals. All states that wait for one of these events in `A` will appear in `C`. If `A` send one of these signals to `B`, a state wait for this event mus appear in `C`. The C will contain all traces (all possible sequence of events) of select signals of `A` and `B` sincronized in it statechart.

The nest law establishes when a behaviour in a capsule can be moved to another capsule which communicate with it.

Rule 13. *Move Behaviour*



provided

(→) *the behaviour always appear immediataly before the capsule* `A` *send a signal* `a` *to capsule* `B`.

(→) *The send of signal* `a` *is always the last action of a state that all out transitions are fired by signals send by* `B`.

The provisos that a case where the two statechart are synchronized. The move of the behaviour preserve the order of actions computations in the system and that the capsule `B` will still free to receive another signal of `A` (don't less the signal) after the cahnges in it statechart. After the move, the behaviour must appear immediatly (the first action) after the receive of signal `a`.

Despite the law pattern show the behaviour as a action, this law can be applied to move a set of state and transions, with respective actions, between capsules. In this case, more ports could be insert in `B` (Law 3), move attributes and methods ((Laws 8 and 9).

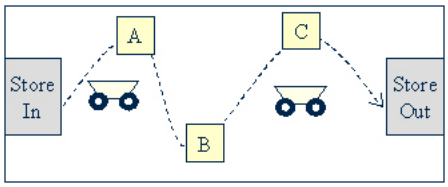
4 Study Case

In order to illustrate the application of the laws proposed in the previous section, we will show a systematic refinement of a abstract representation of a Automatic Manufacturing System, based on the work of Heike Wehrheim [32]. We intend

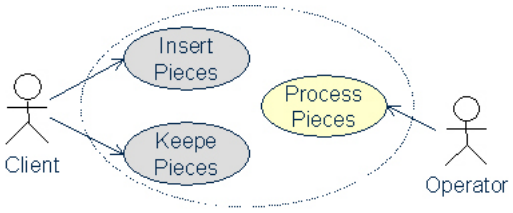
refine con the abstract analysis model, extract of a use case of the system, to more concrete model closer to the specification of Wehrheim [32].

We also motivate us step in activity of the Rational Unify Process (RUP) [33] add to design decision to lead to the Wehrheim specification.

The Automatic Manufacturing System is composed of 2 stores (in and out), three processor (A, B and C) and some robot (called as *Holonic Transportations*). At initial time, the in store are full and the processor with no piece to process. They ask to the robot for peaces that they can process. And the robot must carry the piece between the stores, pass mandatorily to the processor A, after B e last to C.

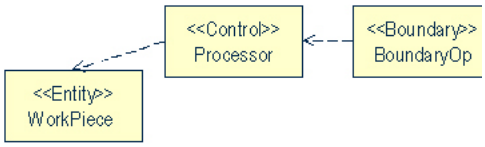


The first step here, after the general vision of the system, is prepare a use case diagram, that leave later to a analysis model, that initially don't have any the final element in the general vision. In the use case, we simplify the system to three functionalities: Insert a Piece; Keep a Piece; and, Process a Piece. A client can insert pieces, and after some time the system processed this he can keep that. The system only initialize the process after a operator start it. In this case study we will exploit only the use case *Process Pieces*.

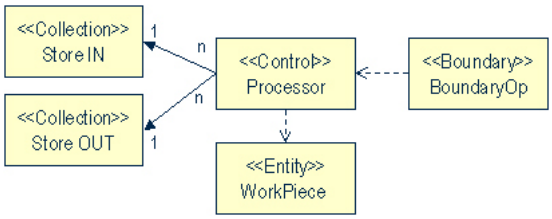


Using some guidelines founded in the RUP, we extract some analysis class. Where Processor represent the control of use case *Process Pieces*. A basic class WorkPiece representing the pieces which will be processed. And a boundary class BoundaryOp which interface the operator action of start the system, creating the processor that make them. We can view this step as create a unique class which all behaviour of the use case, and after this using the refactoring of Fowler *extract class* [6] the create the control a basics classes. For understandability we prefer use in this step the RUP guidelines.

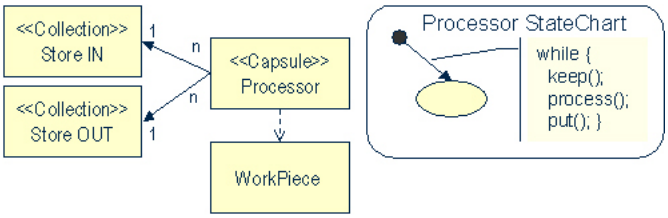
Afterward extract a analysis model, we need found a candidate architecture **??**. In this activity, we identify class that are key to the interaction between



use cases and verify reuses of class. Look at it, we view the need of stores to the processor keep the pieces inserted in the use case *Insert Piece*. Applying the refactoring *extract class* to the control class of each use case, and join then for reusability.

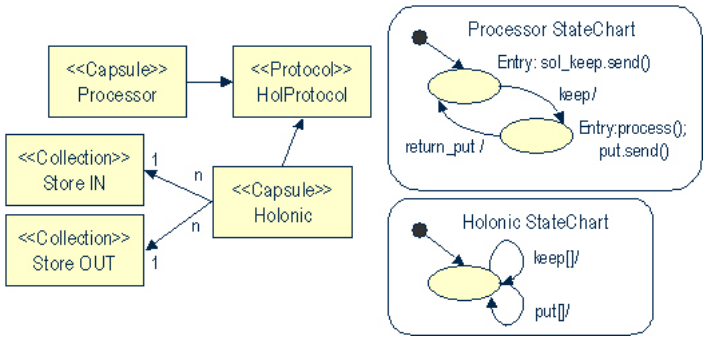


At the activity identify design elements, we start to work with active classes. in this activity are refined the model to better realize the requirements of the system by some design decision. First we observe that Processor need be transformed to a capsule, because they active behaviour of periodic periodically keep pieces to process. Applying the rule 10 we transform Processor in a capsule. Despite we understand that BoundaryOp will be replaced to capsule too, because this react to aperiodicals external events, we not show it here however it not appear in the specification of Wehrheim and is not needed to understand the system, after this point we no show more this element in the diagrams. To create the Processor the BoundaryOp can use Frame service to insert a capsule and connect it to another at run-time.

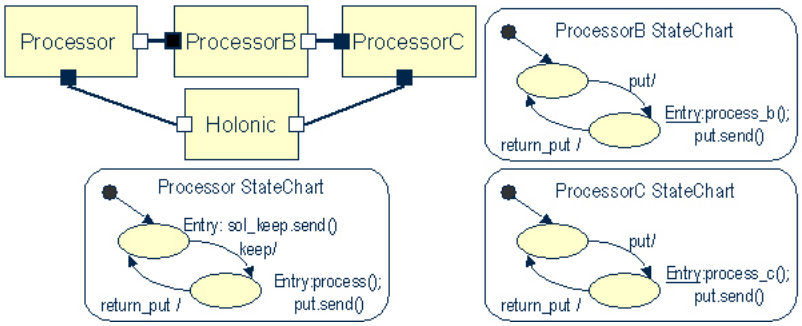


Due to physics requirements, we need insert the holonic transportation (the robot) to keep and put pieces in the stores. To make it, we use the Law 1 to create the Holonic, and after the Law 9 the attributes in and out to there. Now with more that one capsule that interact, we create a live system that send events

between that work without external events. In an future step we can refine the Holonic capsule to put a driver that guide it to locate the stores. Still relate to the physic requirement, we can in a future step put the capsule in different process or threads (define a run-time architecture activity **??**).

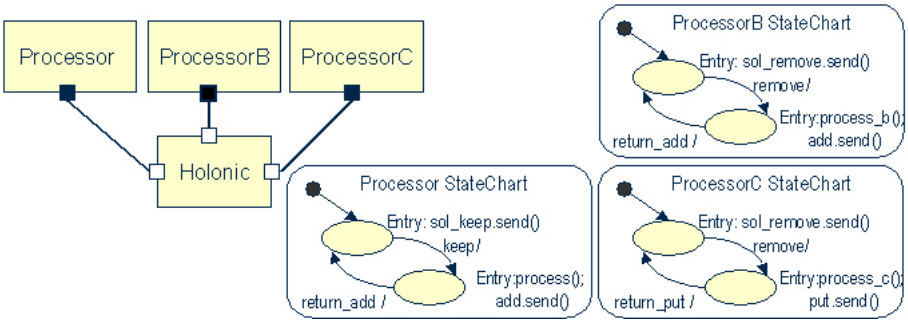


Despite has no activity design Capsule in RUP, we consider this necessary to refine the capsule as we have a activity to refine classes (design class and design use case). Here we will applying a design pattern [**?**] chain of responsibilities, permiting form a pipeline of capsules in the future, improving the performance of the system. To make it, we will use the Law 12, select the signal that put a piece in Holonic, one in `Processor` to create a capsule `ProcessorB` and after it extract a `process_b` method from `process` (Fowler refactoring) and use the Law **??** to move it to `ProcessorB`. After that, use another time the the Law 12 in `ProcessorB` to create a capsule `ProcessorC`, and move a method `process_c` to it.



Another time, for physics requirements we need make the `ProcessorB` and `ProcessorC`, requesting pieces from Holonic we will refine the processors to add some buffers. Starting by `Processor`, before it send a piece to another processor we will put two action, that don't change the behaviour, put the piece in a buffer

and remove a piece of the buffer. Now, we move buffer to `Holonic` (Law 9) and them the action remove to `ProcessB`. After it we prove that if a capsule add a piece in a buffer e another remove after, ignoring the time, will have the same behaviour of directily communicaztion between capsules. Making it to `ProcessorB` and `ProcessorC`, we will have a true pipeline.



Now, we system is very close to founded in work of Wehrheim, but with a unique Holonic. This and others extensions will maked in the future.

5 Conclusions

In many aspects MDE is similar to others unsuccessful approaches in past, failed because the tools couldn't keep up with changing technology or by desired modelling skills of developers, where one example of this approaches are the Integrated Computer Aided Software Engineering (I-CASE) [14]. Today, the community is very enthusiastic with MDE, and working hard on solutions to the what we know did wrong in the 90s and should could be right.

In this work, we proposed many laws for UML-RT which seem useful to support a refinement strategy from analysis to design models, involving mainly diagrams and all elements of UML-RT. Here, the change the occur in the class diagrams, statechart and structure diagram, when we apply each transformation law.

Regarding UML model transformations, we will find several works [34,35,31] that consider only structural diagrams and discard the some behavioural diagrams. The work of Gogolla [34] show some equivalence rules in UML class diagrams. The work of Evans [35] show some refactorings for UML class diagrams, based on a mapping to formal methods. And the work of Grey [31] a set o basic law for refactoring in Alloy, formal language with a semantic more defined and simply than UML.

Another works [36,4] considering others behaviour diagrams in addition to class diagrams, but only transformations seen one diagram per time, without the several correlate changes that occur in the behavioural diagrams when the

trasnformation is applied do class diagrams. The work of Lano [36] gives a formal semantic to UML, OCL and statecharts in term of Real-time Action Logic (RAL). And the work of Sunye [4] relating the concepts of refactoring of Opdyke [7] and Roberts [8] in model transformations, showing the preservation of behaviour in some transformations with statecharts and class diagrams and discussions on equivalence notions.

Relating to transformation in UML-RT existing some work [37,38] on refinements and refactorings in those diagrams, but with a little set of rule and without details about them.

The work in [37] discuss a stepwise development processes with UML-RT exploit two particular refinement principles: Behavioural interface refinement and Incorporating time. The firs issues addressed allow to development interfaces between components step by step, allowing to change the structure of these interfaces together with the statechart specifying its behaviour. In the second issue, Sandner suggest to ignore execution times of component as long as possible, proposing to use asynchronous models at the beginning of development and refine to synchronous model later on.

In the work [38] Engels exploit the locality principle, formalizing the modelling evolution by local transformations, like basic transformations on UML-RT elements (creation, deletion, updating) and to study the effects of these transformations on various consistency properties, focuses in the conserve deadlock freedom and protocol consistency properties.

One immediate topic for future work it make a Normal Formal, that explain all elements of UML-RT on basic UML elements. And thus, proving the complete of set of laws. Others topic involving address the soundness of the laws in detal, with a complete mapping of all notation of UML-RT to OhCircus.

References

1. Thomas, D., Barry, B.M.: Model driven development: the case for domain oriented programming. In: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press (2003) 2–7
2. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20** (2003) 19–25
3. Kent, S.: Model Driven Engineering. In: Proceedings of IFM 2002. LNCS 2335, Springer-Verlag (2002) 286–298
4. Sunyé, G., Pollet, D., Traon, Y.L., Jézéquel, J.M.: Refactoring UML models. In Gogolla, M., Kobryn, C., eds.: UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings. Volume 2185 of LNCS., Springer (2001) 134–148
5. Soley, R., the OMG Staff: Model-driven architecture. Technical report (2000) OMG Document.
6. Fowler, M.: Refactoring-Improving the design of existing code. Addison Wesley (1999)

7. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
8. Roberts, D.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign (1999)
9. Smith, G., Derrick, J.: Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. Formal Methods in Systems Design **18** (2001) 249–284
10. Woodcock, J.C.P., Davies, J.: Using Z-Specification, Refinement, and Proof. Prentice-Hall (1996)
11. Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language User Guide. Addison-Wesley (1999)
12. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International (1985)
13. Morgan, C.: Programming from Specifications. second edn. Prentice Hall (1994)
14. Banker, R.D., Kauffman, R.J.: Reuse and productivity in integrated computer-aided software engineering: an empirical study. MIS Q. **15** (1991) 375–401
15. OMG: Object management group (2004) URL: : http://www.omg.org.
16. OMG/MOF: Meta object facility (mof) specification. Technical report (1997) OMG Document ad/97-08-14.
17. OMG: Uml profile for scheduling, performace and time - request for proposal. Technical report (1999)
18. Douglass, B.P.: Real Time UML - Developing Efficient Objects for Embedded Systems. Addison Wesley (1998)
19. Selic, B., Rumbaugh, J.: Using UML for modeling complex real-time systems. Technical report, ObjecTime Limited (1998)
20. Bézivin, J., Farcet, N., Jézéquel, J.M., Langlois, B., Pollet, D.: Reflective model driven engineering. In Stevens, P., Whittle, J., Booch, G., eds.: UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings. Volume 2863 of LNCS., Springer (2003) 175
21. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture. (2003)
22. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of mda. In: Proceedings of the First International Conference on Graph Transformation, Springer-Verlag (2002) 90–105
23. Richard Paige, J.O.: The single model principle. Journal of Object Technology **1** (2002) 63–81
24. Augusto Sampaio, A.M., Ramos, R.: Class and capsule refinement in uml for real time. 6th Braziliam Workshop on Formal Methods, WMF'03, Campina Grande, Brazil, 2003: Invited Paper (2003) Extended version to appear in in Electronic Notes in Theoretical Computer Science.
25. Woodcock, J.C.P., Cavalcanti, A.L.C.: The Semantics of *Circus*. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: ZB 2002: Formal Specification and Development in Z and B. Volume 2272 of Lecture Notes in Computer Science., Springer-Verlag (2002) 184–203
26. Fischer, C.: Combination and Implementation of Processes and Data: from CSP-OZ to Java. PhD thesis, Fachbereich Informatik Universität Oldenburg (2000)
27. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall (1998)

28. Clark, A., Evans, A.S., Kent, S.: Object-oriented theories for model driven architecture. In: Proceedings of the OOIS MDA workshop,. Number 2426 in LNCS (2002)
29. D'Souza, D., Wills, A.C.: Objects, Components and Frameworks With UML: The Catalysis Approach. Addison-Wesley (1998)
30. Engels, G., Küster, J.M., Heckel, R., Groenewegen, L.: A methodology for specifying and analyzing consistency of object-oriented behavioral models. In: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press (2001) 186–195
31. Gheyi, R., Borba, P.: Refactoring Alloy Specifications. In: WMF 2003: 6th Workshop on Formal Methods, Brazil. (2003)
32. Wehrheim, H.: Specification of an automatic manufacturing system - a case study in using integrated formal methods. In: FASE 2000, Fundamental Approaches to Software Engineering. Volume 1783 of LNCS., Springer (2000)
33. Kruchten, P.: The Rational Unified Process: An Introduction. second edn. Addison-Wesley (2000)
34. Gogolla, M., Richters, M.: Equivalence rules for UML class diagrams. In Bézivin, J., Muller, P.A., eds.: The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998. (1998) 87–96
35. Evans, A.S.: Reasoning with uml class diagrams. In: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques, IEEE Computer Society (1998) 102
36. Lano, K., Bicarregui, J.: Semantics and Transformations for UML Models. In Bézivin, J., Muller, P., eds.: The Unified Modling Language, UML'98 – Beyond the Notation. Volume 1618 of Lecture Notes in Computer Science., Springer-Verlag (1999) 107 – 119
37. Sandner, R.: Developing distributed systems step by step with uml-rt. In Giehse, J., Philippi, S., eds.: Workshop Visuelle Verhaltensmodellierung verteilter und nebenlufiger Software-Systeme, Universitt Mnster (2000)
38. Engels, G., Heckel, R., Küster, J.M., Groenewegen, L.: Consistency-preserving model evolution through transformations. In Jézéquel, J.M., Hussmann, H., Cook, S., eds.: UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings. Volume 2460 of LNCS., Springer (2002) 212–226