Mathematical Model Transformation for System Verification

Dániel Varró and András Pataricza

Budapest University of Technology and Economics
Department of Measurement and Information Systems
1111 Budapest, Hungary,
Pázmány P. st. 1/D. IT building B.420.
Contact person: András Pataricza
Phone: +36-1-463-3595
Fax:+36-1-463-4112
pataric@mit.bme.hu

Abstract The design process of complex, dependable systems requires a precise verification of design decisions during the system modelling phase using formal methods. For that reason, the mathematical models of various formal verification tools are planned to be automatically derived from the system model usually constructed from UML–diagrams. In the paper, a general framework for an automated model transformation system is introduced providing a uniform formal description method of such transformations by applying the powerful computational paradigm of graph transformation.
Keywords: system verification, model transformation, graph transformation, UML.

1 Introduction

1.1 Integrating System and Mathematical Models

For most computer controlled systems, especially dependable, real–time systems with critical applications, an effective design process requires an early validation of design concepts and architectural choices (without wasting time and resources) in order to assess whether the system fulfils its specification or still needs some re-design. This process is called system verification.

The increasing need for effective design has contributed to push for the development of standardized and well-specified design methods and languages, which allow system developers to work on a common platform of design tools.

The Unified Modeling Language (UML) [9] is a general–purpose visual modelling language designed to specify, visualize, construct and document artifacts of a software system. It provides a series of diagrams with the fine level of abstraction to specify object models and has been widely accepted as an object-oriented software design language in the software engineering community.

Formal methods offer a rigorous and effective way to model, design and analyze computer systems. They have been a topic of research for many years with

valuable results, their necessity is vital due to the complexity of IT products and increasing requirements for dependability and Quality of Service (QoS).

During the design phase, a UML designer compares different architectural and structural solutions to select the most suitable one. A mathematical analysis, carried out after modelling (and using formal methods), allows to identify dependability bottlenecks and critical parameters to which the system is highly sensitive.

However, such an integration of system and mathematical modelling (using UML and formal methods) might raise several obstacles:

- At first, mathematical skills are required (at least the basic knowledge of underlying mathematics of a specific method)
- The resources spent on re-modelling are expensive (in time and workload).
- How can the faithfulness of the model be guaranteed?
- How can the consistency of the model and the original problem can be checked?
- How can the results of the verification be back-annotated to the system model?

1.2 Mathematical Model Transformation and its Problems

Several sophisticated tools based on formal methods (such as [6]) are available for analyzing mathematical models constructed from a UML-based system model. The step when the input language of these mathematical tools are generated from the UML model of the system is called **mathematical model transformation**.

The inverse direction of model transformation (referred as **back–annotation**) is of immense importance as well when some sort of problems (e.g. a deadlock) arise during the mathematical analysis. After an automated back–annotation these problems may appear in the UML system model level allowing the designer to fix these conceptual bugs.

Several semi–formal approaches have been designed and implemented performing mathematical model transformation in our sense.

- A transformation has been defined which maps a subset of UML Statechart Diagrams to Kripke structures for formal verification of functional properties [7] using SPIN model checker [6]
- A projection from UML subsets to timed and generalized stochastic Petri Nets ([4, 3]) have been defined for providing quantitative analysis of dependability attributes.

However, these semi–formally/informally defined transformation algorithms raised several problems due to the lack of a uniform formal description method and a straightforward strategy for implementation.

- There was a gap between theory and practice as the strategies of implementation were rather ad-hoc, thus requiring months of related work on implementational issues.

– Moreover, the transformation scripts were written in PL/SQL, which is highly data-dependent, hence their formal verification (aiming to prove correctness and completeness) was almost impossible.
– Each model and model transformation had to be verified individually although the transformation algorithms have similar underlying algorithmic skeletons.

1.3 Research Objectives

The aim of our research is to provide a framework for a general mathematical model transformation system supporting the automated generation of transformation code of a proven quality [11].

Such **an automated model transformation system** has to support at least the following features:

1. **The description of source and target models** (input and output);
2. **Th description of the transformation**
3. **An efficient back-annotation** of mathematical analysis results;
4. **A database for storing models and rules**;
5. A transformation engine built upon this database providing **automatically generated transformation code**;
6. **An engine for proving correctness and completeness**;

In the current paper, we basicly focus on Requirement 2, and 3, although, an overview of the entire architecture and concept is provided as well in the following.

1.4 Visual Automated Graph Transformation System

Our alternate solution for mathematical model transformation (depicted in Figure 1 and 2) is an integration of different disciplines of computer science, computer engineering and artificial intelligence. Based on formal mathematical background, it provides a general transformation description method and a proposed way of designing such transformations, additionally.

User–created system models are defined by the **Unified Modeling Language**, which is the front–end of most model transformations. UML conceptually follows the four–layer **MOF** meta-modelling architecture [8], which allows the definition of meta–objects for similarly behaving instances.

The front–end and back–end of a transformation (UML as the source model and a formal verification tool as the target model) is defined by a uniform, standardized description language of system modelling, that is, **XMI (XML Metadata Interchange)**. XMI documents are also used for storing the transformation rules.

The formal description of these transformations are supported by **graph transformation**, which combines the advantages of graphs and rules into an individual computational paradigm. Both transformation rules (denoted as **transformation rule description (TRD)** in Figure 1) and source and target side grammar rules are given in a special form of graph transformation rules.

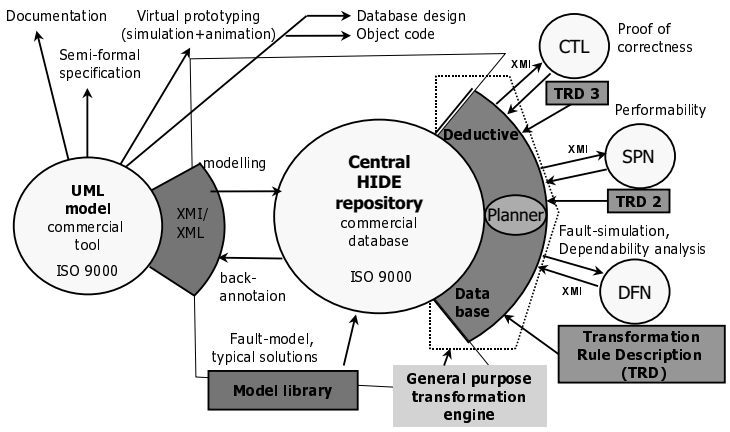


Figure 1. Our proposed model transformation architecture

A **graph transformation rule** is a special pair of pattern graphs where the instance defined by the left hand side is substituted with the instance defined by the right hand side when applying such a rule (similarly to the well–known grammar rules of Chomsky in computational linguistics). A collection of such rules forms a **graph grammar** for describing system and mathematical models as **visual languages**. An efficient back–annotation of analysis results is also supported by a special relation between source and target graph objects.

After having specified a set of transformation rules, typically, two main problems arise for obtaining a higher quality of transformation code:

- **Correctness (soundness) problem**: Whether this set of rules is (syntactically) correct, i.e. the output is a well–formed sentence of the target language.
- **Completeness problem** Whether this set of rules is complete, i.e., each situation allowed by the source language is covered by a corresponding rule [11].

With the supervision of **planner algorithms** of artificial intelligence, both problems can be verified constructively (i.e. pointing out those parts of the system where correctness or completeness do not hold). As a result, there is an intensive increase in the reliability of transformation as only the design of elementary rules needs intuition and the deep relationship between them is automatically generated.

Source and target models are stored in a central repository, which is a relational database. However, such complex transformations necessitate more intelligent query processing algorithms than original SQL queries. Therefore a

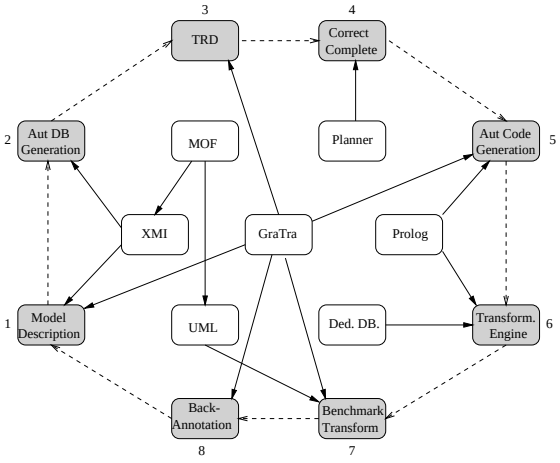


Figure 2. An overview of model transformation

deductive database was carried out in Prolog to provide a higher level query language for describing complex database operations in our transformation engine.

Even if the description of the transformation is theoretically correct and complete, additionally, the source and target models are also mathematically precise, the implementation of these transformations has a high risk in the overall quality of a transformation system. As a possible solution, **automatic transformation code** and **database generation** is aimed based on XMI documents and visual transformation rules.

Our design concepts are demonstrated on a **benchmark transformation** projecting structural UML diagrams into timed Petri Nets [4, 3]. The specification of the transformation, which form the rest of the current paper, is defined by the precise formalism of **transformation units** (of graph transformation).

The rest of the paper is structured as follows. In Section 2, some basic notations of graph transformation will be given. Section

2 Concepts of Graph Transformation

Graphs are well–known and frequently used means to represent complex objects, diagrams and networks, like flowcharts, entity-relationship diagrams, Petri Nets, and many more. Rules have proved to be extremely useful for describing local transformations; areas like e.g. language definition, logic and functional programming, theorem proving.

Graph transformation combines the advantages of both graphs and rules into a single computational paradigm, used for generation, manipulation, recognition, and evaluation of graphs [1].

In the sequel, some basic definitions of graph transformation systems are given. Section 3 contains a short introduction of our benchmark transformation from UML diagrams to timed Petri Nets. In Section 4, a subtransformation of the previous will be discussed in details on a running example, finally, Section 5 concludes our paper.

2.1 Basic Notation

Definition 1. *A **graph transformation rule** $r = (L, R, App$ consists of a left–hand side (LHS) graph L, a right–hand side (RHS) graph R, and application conditions App.*

A graph transformation rule p can be applied to a given *host* graph G by the following procedure: [5]

1. Find an *occurrence* of the left–hand side L in G (satisfying application conditions App).
2. Remove a part of graph G determined by the occurrence of L, yielding the *context* graph D.
3. Embed the right–hand side R into D, obtaining the *derived* graph H.

Definition 2. *A **model transformation rule** $mtr = (LS, LT, RS, RT, App)$ is a special graph transformation rule, where both L and R graphs can be divided into a **source** and a **target** graph (LS, LT, RS, RT respectively) connected by **reference** relations.*

Definition 3. *A **transformation unit** $tu = (I, U, R, C, T)$ encapsulates a specification of initial graphs I, a set of imported transformation units U, a set of graph transformation rules R, a control condition C, and a specification of terminal graphs T.*

The operational semantics of transformation unit is a binary relation on graphs, which relation contains a pair (G, G') of graphs [1], if

- G is an initial graph and G' is a terminal graph,
- G' can be obtained from G by derivations of own rules and imported transformation units.
- The pair and the derivation process is allowed by the control condition.

Although a graph transformation rule is non–deterministic in general, our model transformation mechanism is restricted by control conditions prescribing a specific (deterministic) order of applying rules.

In the current paper (when examples are presented later in Section 4), graph (model) transformation rules will be displayed in a pictorial form, while transformation units will be defined textually. As a graph transformation rule may consist of several layers such as a graphical and a logical layer (e.g. in [2]), using only the graphical notation is still sufficient for specifying transformations.

3 From Structural UML Diagrams to Timed Petri Nets

In Section 3, a benchmark transformation of our model transformation system will be discussed. The transformation itself (projecting structural UML diagrams into Timed Petri Nets) was introduced in [3, 4] for assessing and evaluating dependability parameters of the system in an early phase of the design process.

Such an automated transformation from UML structural diagrams to timed Petri Nets serves as (the informal specification and purpose of the transformation basicly follows [3, 4]):

- to provide means to analyse dependability attributes of a system still under design.
- to allow a less detailed, but system–wide representation of the dependability characteristics of the analysed system.
- to deal with various level of details, ranging from preliminary abstract UML descriptions, up to the refined specifications of the last design phase.

3.1 Outline of the Transformation

The main approach when preparing a formal analysis of UML models is to enrich the basic UML models by different attributes needed for analysis. Obviously, these extensions, which have to fit into the syntax of UML descriptions, are only partially relevant in a particular analysis method. In our case, performance parameters associated with each class or instance have to be transformed into a stochastic Petri Net model built up from the basic model by an automatic transformation in order to analyze the performance issues.

The entire transformation from structural UML diagrams to timed Petri Nets is performed in two steps as depicted in Figure 3.

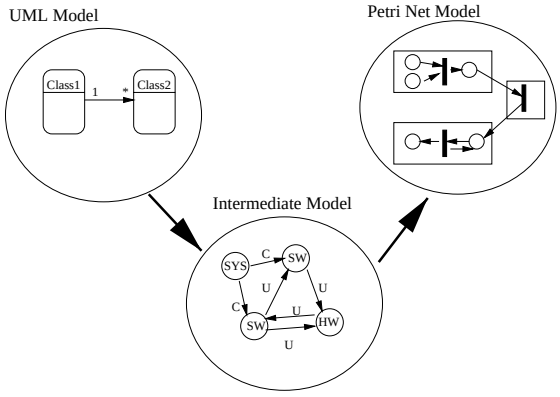


Figure 3. The outline of the UML—Petri Net transformation

1. The first task is to extract the relevant dependability information from the mass of information available in the UML description, resulting in the **Intermediate Model (IM)** of the system. In this step, a dependability model of the system is constructed focusing on the basic events, error propagation paths, etc.
2. The next step allows to define a timed Petri Net description general enough to postpone the choice of a specific tool used for the analysis to a later stage.

Modelling redundancy structures in the **UML–based system model** is defined as instances of specially stereotyped classes. These stereotyped classes allow the designer to attach tagged values required for calculating dependability measures. Different levels of abstraction are expressed by the use of use cases, classes, components and objects.

The **Intermediate Model** of the system is a special **hypergraph** with various types of attributed nodes and edges. Both node and edge types are mainly based on the UML stereotypes of their corresponding instances. Once having constructed the IM from the UML system model, we need not use the original system for the rest of the transformation process any more.

The generated **Petri Net (PN) model** is hierarchically constructed from partially connected PN subnets (for each type of IM nodes). These subnets are linked via propagation subnets (the transformed equivalents of hyperedges). In such a way, a structural decomposition of Petri Nets can be obtained.

Although the entire transformation is divided into two subtransformations (UML to IM and IM to Petri Net), only the first part of the transformation is discussed in the current paper on a running example due to the following reasons:

- The complexity of transformation rules originates in the complexity of the source language. Hence, the more complex UML as the source language has been chosen for demonstration purposes rather than the more simple hypergraph structure of the IM.
- Dealing with the structural problems of UML is reusable in the implementation of further model transformation.
- As a single IM graph node is often transformed into a Petri subnet of connected places and transition, the RHSs of the translation rules are often two large to be easily depicted.

In the current paper, dependability parameters are omitted , since the complexity of model transformations originates in the *structure* of the models. As typed and attributed graphs are usually used for graph transformation, these parameters can easily be handled as described e.g. in [10].

3.2 Our Running Example: the Production Cell

Our running example (taken from [4] and depicted in Figure 4) models a production cell, which has been adopted in the literature as a benchmark for modelling reactive systems.

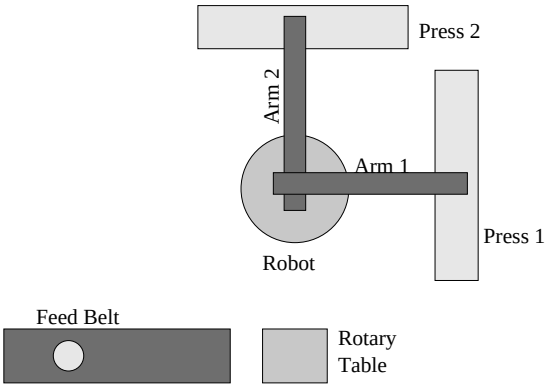


Figure 4. The production cell

A production cell processes metal plates taken to the cell by a worker. A plate is conveyed to a rotary table on a feed belt. The rotary table is used to move the plate to a position that is proper for a robot to take the plate, and place it into a press. The press forges the plate, which is then removed by the robot and given back to the worker. The various elements of the production cell are controlled by software modules running on a single PC. In order to tolerate the failure of a press, two redundant presses are used.

3.3 UML Design of the Production Cell

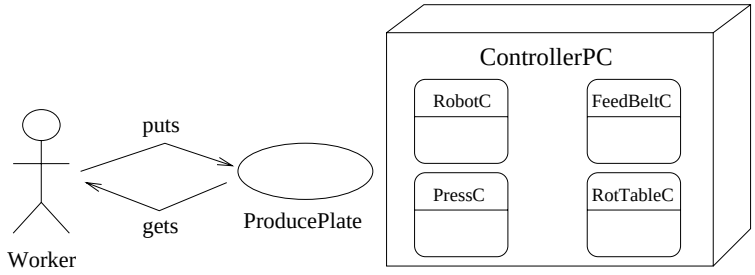
Use case diagrams are used to collect high–level system functionalities (use cases), and participants (actors) interacting with the system by using these functions. In the sense of dependability, use cases represent the top level service of the system this way also defining the system level failure.

Figure 5(a) shows us a simple use case diagram for the production cell. The actor `Worker` uses the `ProducePlate` functionality of the production cell system (depicted without stereotypes).

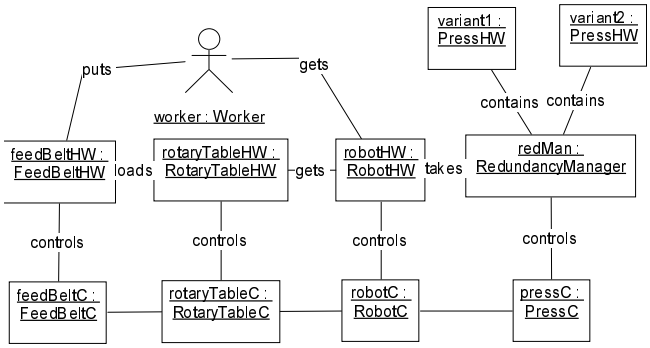
The production cell is modelled by a set of objects (depicted in the **object diagram** of Figure 5(c)), each representing either a hardware unit (e.g `RotaryTableHW` is the rotary table, including its sensors and actuators) or part of the controller system (e.g. `FeedBeltC` is a piece of the controller software responsible for the feed belt). Object `Worker` is also included to show the interaction with the environment.

Links between the objects have the following meaning:

- the machines have states and operations, which are set and sensed by the control software
- the software components cooperate to control the safe and efficient operation of the cell and



(a) Use case diagram (b) Deployment diagram



(c) Object diagram

Figure 5. UML design of the production cell

– the machines interact by performing operations on a plate.

In order to tolerate the failure of a press, two redundant presses are used in the cell. A separate class of objects (`RedundancyManager`) is used to perform the task of selecting the available press (indicated by `variant` stereotypes) and forwarding the control to it, this way the pure functional control can be performed by the same object `PressC`.

The control software classes are deployed on a single PC as defined by the **deployment diagram** of Figure 5(b) (naturally, further hardware objects are also deployed on various nodes).

3.4 The Intermediate Model of the Production Cell

The Intermediate Model of the system collects the relevant entities, and relations projected from the UML system model.

The Intermediate Model (a well–formed hypergraph) of the basic production cell is depicted in Figure 6.

The following nodes and relations (hyperedges) are used.

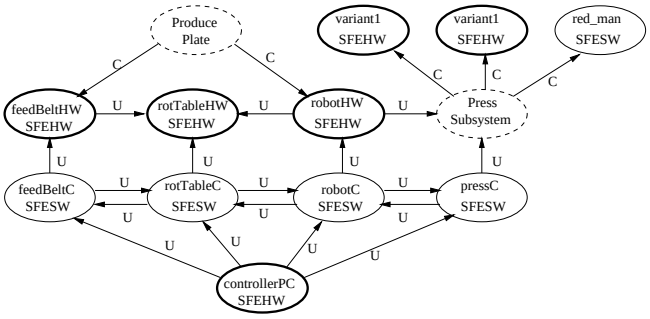


Figure 6. Intermediate model of the production cell

- The main service of the system is represented by a `SYS` node (drawn using dashed lines). The relations (hyperedges) of type `C` (composed of) identify the nodes that represent objects used directly by the worker. The failure of the system can be recognized when the feed belt or the robot provides improper service (the plate is not taken or no plate is returned).
- The components of the system are represented either by stateful hardware (`SFEHW`, drawn using thick lines) or stateful software nodes (`SFESW`, drawn by thin lines).
- The links among objects are represented by hyperedges of type `U` (uses the service of).
- The deployment of the software is projected to a set of unidirectional `U` edges.
- The redundancy structure is identified on the basis of the stereotypes. In the IM, this structure is represented by using `FTS` type nodes (using dashed lines again) and `C` relations.

4 The UML—Intermediate Model Transformation

The specification of the transformation will be presented at two layers of abstraction.

1. Starting from a **semi–formal description** (as defined in [3]), the basic concepts of a local transformation (e.g. transforming use cases) will be introduced.
2. A **formal semantics** of the UML–IM transformation will be given by means of **transformation units** (see Section 2) composed of model transformation rules. Transformation units are listed in a top–down order, starting from the most general one.

The main transformation unit `transform_UML2IM` is responsible for the entire model transformation. A transformation unit is listed in a framebox with its name and arguments on the top of the box. Its components (initial and terminal graph, rules, imported units and control condition) are printed in italics

(however, empty components will be omitted later). All the imported units and graph transformation rules are defined sooner or later in the paper, however, only non–empty components will be explicitly printed.

transform_UML2IM(SG,TG):

initial: $TG = \emptyset$
rules:
import:
$transform\_elementTU(SG,TG), redundancy\_structureTU(SG,TG),$
$transform\_relationTU(SG,TG),$
condition:
$transform\_elementTU(SG,TG); redundancy\_structureTU(SG,TG);$
$transform\_relationTU(SG,TG);$
terminal: *once all occurrences in the given order*

According to the basic transformation unit (`transform_UML2IM` with two attributes `SG` and `TG`), three local subtransformations have to be performed in the given order, namely,

- projecting hardware and software element (`transform_elementTU`),
- transforming redundancy structures (`redundancy_structureTU`)
- projecting relations (`transform_relationTU`), and

At this point, no "own" (owned by unit `transform_UML2IM`) transformation rules are available (`rule:` section is empty), all operations are imported.

The initial condition of transformation unit (`transform_UML2IM`) prescribes that the target graph TG should be empty while no restrictions are required for the source graph SG. A terminal graph is obtained if all matching occurrences are transformed once in the specified order.

In our transformation units, control conditions (composed of transformation rules and units and sequence operators in our case) always determine the priorities (by prescribing the entire sequence, separated by semi–colons as sequence operators) of transformation units, therefore the units have to be applied in the given, deterministic order. Due to the deterministic completion and the guaranteed termination, the definition of terminal graphs is not required.

4.1 Projecting Hardware and Software Elements

transform_elementTU(SG,TG):

import: $usecaseTU(SG,TG), nodeTU(SG,TG), objectTU(SG,TG).$
condition: $usecaseTU(SG,TG); nodeTU(SG,TG); objectTU(SG,TG).$
terminal: *once all occurrences in the given order*

The projection of hardware and software elements includes the) handling of the following UML diagrams in a similar way:

- use case diagrams
- deployment diagrams.

– object diagrams

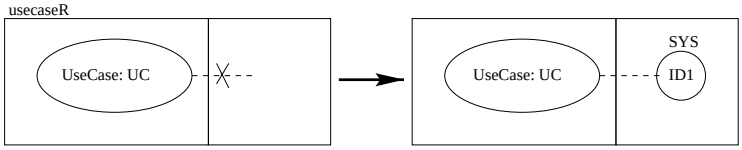
At this point, rule ordering could have been arbitrary (e.g. transforming nodes before objects could also be a suitable order) due to the lack of casual links between these transformation units. Therefore, considering parallel completion (independent threads for rules/units of same priority) instead of strict serialization is worthwhile for future works.

The role of **use case diagrams** is to identify system level relations by relations between actors and use case.

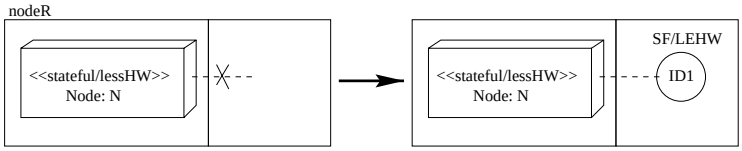
– A **use case** without a stereotype represents a refined use case (relevant for the current analysis), which is projected into a system node of the IM.
– **Actors** represent users, or external entities which interact directly with the system, hence not projected in this transformation.

usecaseTU(SG, TG):

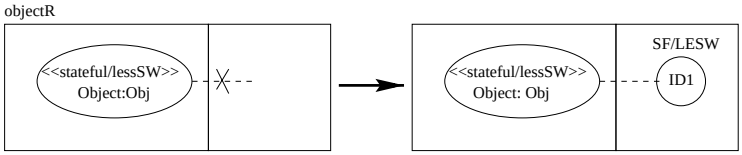
initial: $usecase(UC), \neg transformed(UC, UCT)$
rules: $usecaseR(UC, SG, TG)$ (Figure 7(a))
condition: $usecaseR(UC, SG, TG)$
terminal: *once all occurrences in the given order*



(a) Transformation rule of use cases



(b) Transformation rule of HW nodes



(c) Transformation rule of objects

Figure 7. Transformation rules of use cases, HW and SW elements

For graph transformation rules in the paper, the following graphical notation is used. Graph nodes of the source model (UML) are depicted as their corresponding UML concept, while the hypergraph of the targe IM model is drawn like an ordinary graph with labelled circles (as nodes) and directed arrows (as edges). Reference relations (supporting an efficient back–annotation) are dashed lines, and negative reference condition (i.e. a missing reference relations) is depicted by a crossed dashed line.

The transformation rule `usecaseR` indicates that a use case that is not transformed yet, indicated by a negative condition (the crossed reference line) without a stereotype should be transformed into an IM node with type `SYS` and a reference relation is also inserted between the source and target elements (namely, the use case and the IM node). The result of applying the transformation unit is displayed later in Figure 10(a).

Nodes are run–time physical objects, usually hardware resources. They are projected into hardware elements in the IM.

nodeTU(SG,TG):

initial: $node(N), \neg transformed(N, NT)$
rules: $nodeR(N, SG, TG)$ (Figure 7(b))
condition: $nodeR(N, SG, TG)$
terminal: *once all occurrences in the given order*

The transformation rule `nodeR` (and `objectR` later on) performs a similar task to `usecaseR`, creating a new IM node and linking it with a reference relation.

An **object** is a particular instance of a class. It has identity and its own state therefore it is projected into a software element of the IM as depicted in Figure 7(c).

objectTU(SG,TG):

initial: $object(Obj)\ \neg transformed(Obj, ObjT)$
rules: $objectR(Obj, ST, SG, TG)$ (Figure 7(c))
condition: $objectR(Obj, ST, SG, TG)$
terminal: *once all occurrences in the given order*

4.2 The Projection of Relations

Although the transformation of redundancy structures preceed the projection of relations, the latter one is discussed first.

In dependability modelling, these relations and links are supposed to indicate potential error–propagation paths between model elements. These relations may appear in different UML diagrams (the list contains only those that considered in our running example):

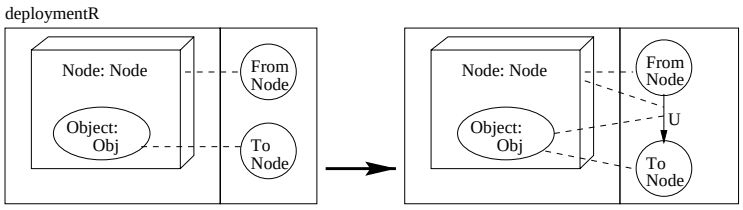
- relations among use cases and actors in use case diagrams,
- links (as instances of associations or aggregations) between objects
- component containment in deployment diagrams

transform_relationTU(SG,TG):

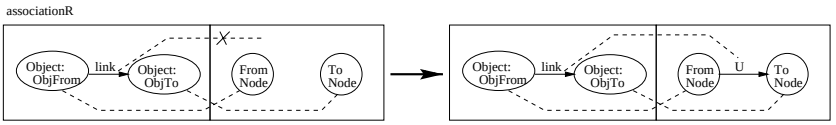
import: $deploymentTU(SG,TG), linkTU(SG,TG), systemTU(SG,TG)$
condition:
$deploymentTU(SG,TG); linkTU(SG,TG); systemTU(SG,TG)$
terminal: *once all occurrences in the given order*

Generally, uni-directional links (like e.g. aggregation) will be projected to a single hypergraph edge in the IM, while bi–directional links (like associations with navigability at both association end) will be projected into two edges with opposite direction.

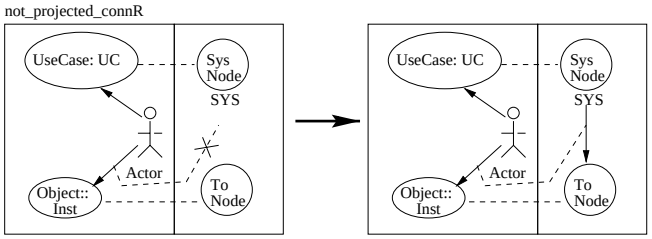
Deployment relations among nodes and components, and relations among components and objects (shown by graphical nesting) indicate potential error propagation paths with direction from the nodes to the objects.



(a) Transformation rule of deployment



(b) Transformation rule of links



(c) Transformation rule of system nodes

Figure 8. Transforming UML relations

deploymentTU(SG,TG):

rules: *deploymentR*(SG,TG) (Figure 8(a))
condition: *deploymentR*(SG,TG)
terminal: *once all occurrences in the given order*

The `deploymentR` transformation rule searches for a hardware node containing a class with an instance as source pattern, and the corresponding IM nodes of the instance and the hardware node as target pattern, linked by reference relations. As a result, an additional IM edge of type U (uses the service of) is created, and linked to both the component class and the hardware node as reference.

Links are binary relation instances (of associations) between objects. In general, links mean that the objects instantiated from the corresponding classes know about each other. A link indicates a potential bidirectional error propagation path between these objects, thus it is projected to the IM.

linkTU(SG,TG):

rules: *linkR*(SG,TG) (Figure 8(b))
condition: *linkR*(SG,TG)
terminal: *once all occurrences in the given order*

The rule `linkR` has a complex LHS. The specific objects within a link has to be found with a navigable link end as the source pattern, and the reference IM nodes of these objects as target patterns. The transformation results in an additional IM edge between the corresponding target IM nodes with a reference relation to both the corresponding link and link end.

A **relation between an actor and a use case** (Figure 8(c)) is not directly projected. Although each relations in the context of a use case without stereotype, are projected from the IM `SYS` node to the corresponding IM node. This case is illustrated by the IM edges inherited from the Worker–FeedBeltHW relation (see Figure 6).

systemTU(SG,TG):

rules: *UCrelR*(SG,TG) (Figure 8(c))
condition: *UCrelR*(SG,TG)
terminal: *once all occurrences in the given order*

4.3 The Projection of Redundancy Structures

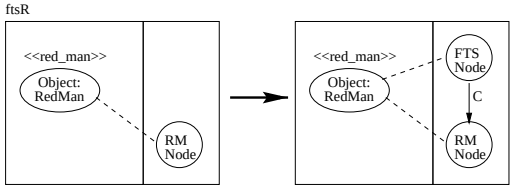
Redundancy structures are identified by stereotyped objects: the redundancy manager (stereotyped as `red_man`) and the variants (with `variant` stereotypes).

- Up to now, the redundancy manager (as being an ordinary object as well) has already been transformed among other objects.
- The variant instances have also been projected into the IM (together with other ordinary objects).
- Additionally, the fault tolerant structure as a whole are projected into an IM element typed `FTS`. This element is connected to the elements representing

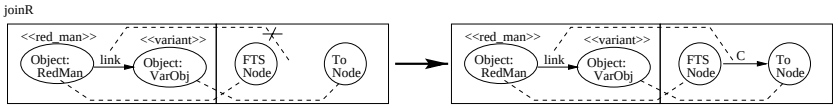
the redundancy structure, and the variants using special "is composed of" relations to represent the components as a whole.

redundancy_structureTU(SG,TG):

import: $redundancy\_managerTU(SG,TG), joinTU(SG,TG)$
condition: $redundancy\_managerTU(SG,TG); joinTU(SG,TG)$
terminal: *once all occurrences in the given order*



(a) Transforming redundancy managers



(b) Transforming links of redundancy managers and variants

Figure 9. Transforming redundancy structures

redundancy_managerTU(SG,TG):

rules: $ftsR(SG,TG)$ (Figure 9(a))
condition: $ftsR(SG,TG)$
terminal: *once all occurrences in the given order*

The `ftsR` transformation rule connects an additional `FTS` type IM node to each redundancy manager, which has two reference IM nodes from this point.

joinTU(SG,TG):

rules: $joinR(SG,TG)$ (Figure 9(b))
condition: $joinR(SG,TG)$
terminal: *once all occurrences in the given order*

Transformation rule `joinR` contains specially stereotyped variant objects linked to the redundancy manager as source pattern, and `FTS` together with `SFEHW` nodes on the target side. As a result, the `FTS` node representing the redundancy structure as a whole is connected to the corresponding IM node of variant objects.

4.4 Tracing the UML—IM Transformation

Finally, a sample run of the transformation algorithm will be traced to observe the sequence in which the objects of the target language are created one by one.

A series of figures showing the construction of the IM (of Figure 6) will be presented with the following notation:

- An IM node is coloured to dark grey if it has been transformed by the application of the latest rule.
- An IM node is painted to light grey in case it has been projected previously.
- An IM edge appears in thick dashed lines if it has been transformed by the last applied rule.
- An IM edge appears in thick normal lines if it has previously been transformed.

Step 1: Transforming HW and SW nodes (Figure 10(a), 10(b) and 10(c))

1. Transforming use case objects by applying transformation unit `usecaseTU`.
2. Projecting hardware nodes by applying `nodeTU`.
3. Transforming all the objects using `objectTU`.

Step 2: Transforming redundancy structures (Figure 10(d) and 10(e))

1. Creating `FTS` nodes and connecting them to redundancy managers by `redundancy_managerTU`.
2. Connecting variants and `FTS` nodes using `joinTU`.

Step 3: Transforming relations (Figure 10(f), 10(g), and 10(h))

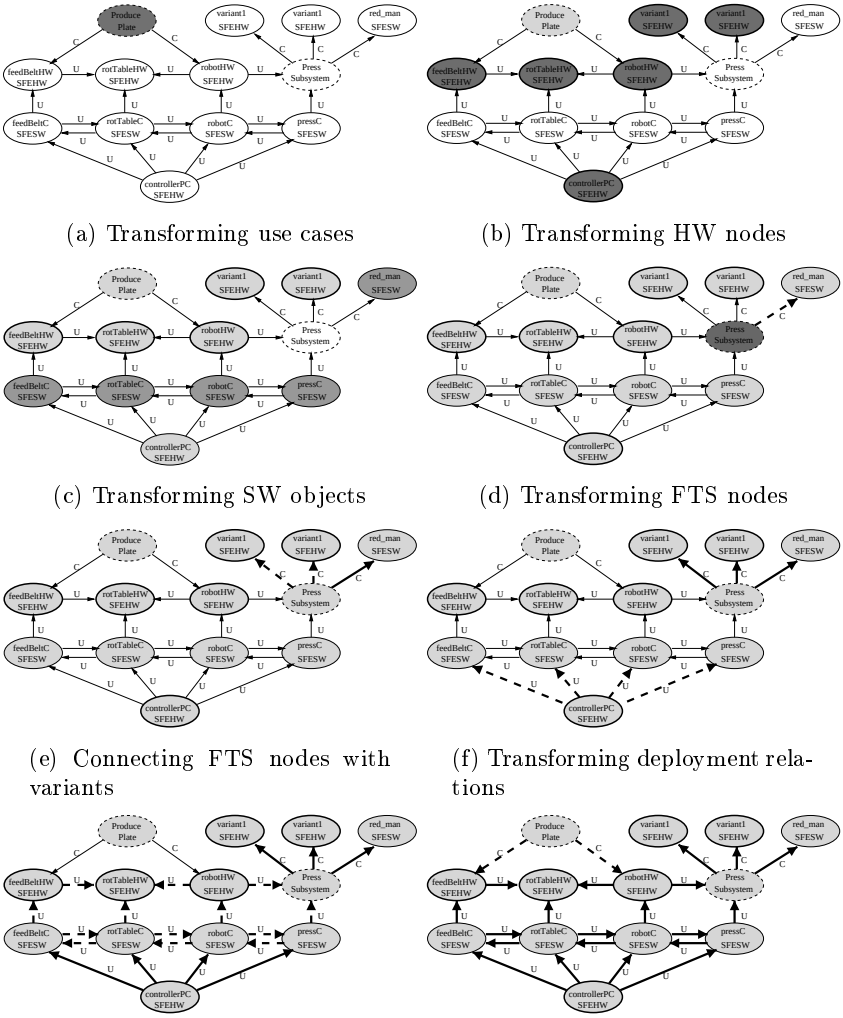
1. Transforming deployment relations by `deploymentTU`.
2. Using `linkTU` for projecting links of objects.
3. Transforming use case relations by applying `systemTU`.

5 Conclusion

In the current paper, our research activities towards a visual automated model transformation system was introduced. Our proposal is supposed to integrate UML–based system modelling and formal methods by providing a general description method based upon visual graph transformation rules.

To demonstrate the expressive power of these rules and transformation units, a formal description of a complex transformation from structural UML models to timed Petri Nets (introduced semi–formally in [4, 3]) was presented on an example (which is a benchmark of modelling reactive systems). Formal specification of further complex transformations (between UML models and different tools of formal methods) will also be aimed in future works.

Although omitted from current issue, the UML–IM transformation has already been implemented using our Prolog–based deductive database implementation. The transformation code strictly follows the control conditions and model transformation rules of transformation units defined earlier in Section 4. In this respect, our proposal is much more closer to an automatic transformation code generation from a formal specification than any previous solutions on the same topic.



(a) Transforming use cases

(b) Transforming HW nodes

(c) Transforming SW objects

(d) Transforming FTS nodes

(e) Connecting FTS nodes with variants

(f) Transforming deployment relations

(g) Transforming links

(h) Transforming use case relations

Figure 10. Tracing the transformation step by step

References

1. M. Andries et al. Graph transformation for specification and programming. *Science of Computer Programming*, (34):1–54, 1999.
2. R. Bardohl and G. Taentzer. Defining visual languages by algebraic specification techniques and graph grammars. In *IEEE Workshop on Theory of Visual Languages*, 1997.
3. A. Bondavalli, I. Majzik, and I. Mura. From structural UML diagrams to Timed Petri Nets. *PDCC-CNUCE European ESPRIT Project, HIDE Deliverable 2, Section 4*, November 1998.
4. A. Bondavalli, I. Majzik, and I. Mura. Automatic dependability analyses for supporting design decisions in UML. *HASE'99: the 4th IEEE International Symposium on High Assurance Systems Engineering*, 1999.
5. A. Corradini and H.-J. Kreowski. GETGRATS and APPLIGRAPH: Theory and applications of graph transformation. In H. Ehrig and G. Taentzer, editors, *GRATRA 2000, Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 283–288, March 2000.
6. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.
7. D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML Statechart Diagrams. In *Proc. IFIP TC6/WG6.1 3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, 1999.
8. Object Management Group. *Meta Object Facility Version 1.3*, September 1999. http://www.omg.org.
9. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
10. D. Varró. Automatic transformation of UML models. Master's thesis, Budapest University of Technology and Economics, 2000. http://domino.inf.mit.bme.hu/biblio.nsf ???
11. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. In H. Ehrig and G. Taentzer, editors, *GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 14–21. Technical University of Berlin, Germany, March 2000.