Model Transformation in Practice

Ralf Gerlich, Daniel Sigg, Rainer Gerlich

*BSSE System and Software Engineering, Auf dem Ruhbuehl 181,
88090 Immenstaad, Germany, Phone +49/7545/91.12.58, Mobile +49/171/80.20.659,
Fax +49/7545/91.12.40, e-mail:Ralf.Gerlich@bsse.biz, Rainer.Gerlich@bsse.biz, Daniel.Sigg@bsse.biz,
URL: http://www.bsse.biz*

ABSTRACT:

The intention of this paper is to highlight the benefits of model exchange between different tools, methods and notations on one side, and to identify issues of proper modelling on the other side which have been detected during model transformation and code generation from models.

Firstly, model transformation is applied to achieve diversification of tools, i.e. to enable the use of different tools on the same model. Equivalent models are derived from one original model and the output from at least two tools is compared. This increases the chance of detecting a fault which one of the tools did not flag during verification or may have introduced during code generation. Secondly, verification and validation of models and the generated code do benefit from different, complementary tool capabilities. A tool may only support a certain set of verification and code generation issues, while other ones may support the missing ones.

The following model notations were considered: UML2 [1], 3ADL [2] and ISGL [3]. Transformations were established from UML2 and 3ADL to ISGL.

For the models received in one of the notations UML2 or 3ADL, the contents was checked on feasibility of automated code generation when transforming to ISGL regarding a fully deterministic and automated transformation into executable code. A number of issues were identified which compromise or prevent unambiguous code generation from UML2 and 3ADL models.

The benefits of diversification were confirmed by the performed activities. Due to the automated transition to another tool more faults could be identified. In addition, it could be proven that modelling faults can be identified on the level of the generated code due to auto-coding. This allows to check and evaluate modelling properties in the execution environment under real conditions rather than in an isolated modelling environment where real conditions are replaced by assumptions and hypotheses.

1 INTRODUCTION

Models raise the level of abstraction of software development towards the application, thereby allowing to express properties in an adequate manner without being enforced to think about implementation details. Today, a variety of modelling languages and supporting tools exist which target different issues of modelling and verification and validation of a model.

Therefore, it should be interesting to compare tools and exploit their capabilities. Though it can be assumed that every tool vendor does his very best on the quality of a tool, from a rigorous point of view it must be doubted that the outcome of verification, validation and code generation is always fully correct.

As the location of a fault not detected during verification or introduced by code generation is not known, an efficient way to identify such faults is to compare the results from different tools, provided that they are really independent.

When transforming one notation into another one, both notations are compared. From a principal point of view, a transformation is only possible, when the intersection of both notations is not empty. Moreover, only such parts of a model can be transformed which belong to the intersection. Otherwise, an assumption has to be made to substitute required, but missing information, which however may compromise the validity of the model.

In practice, it turned out that more modelling elements were outside the intersection than expected. This even appeared in cases where the modelling elements were considered as equivalent.

In particular, the transformation of a model into source and executable code enforces resolving of implicit assumptions and ambiguities. It was recognised that after code generation from a model the behaviour of the generated code may differ among different tools, depending on the tool used and the assumptions implicitly used in the tool to resolve ambiguities and to substitute missing information.

From a verification and validation (V&V) point of view the properties of the generated code are the essential ones, and these may not comply with what is expected when defining a model and verifying it at modelling level. To ensure compliance with the generated code, all desired properties of the final product need to be defined in a model based on the used notation. However, this important requirement is not fulfilled in most cases.

This leads to the following conclusions: when the desired property can be observed on the (automatically) generated code, the code is acceptable w.r.t to the desired property. If a desired property cannot be observed, it has to be checked whether this corresponds to a fault in the model – implying unsuccessful V&V at modelling level, or in the code generation process.

In case of a transformation from UML2 it could be proven, that the faults were present in the model when observing faults in the generated code. The reason simply is that the tool by which the model was established does not allow identification of such faults. This clearly identifies the benefit of tool diversification.

A further conclusion is: the success of V&V activities is only valid in the context of the applied V&V capabilities of the tool, no general conclusion on the correctness of a model should be drawn, unless all of the desired properties have been confirmed on the level of the generated code.

In the context of the executed V&V activities the following types of faults were found by model transformation: dead code on modelling level, performance issues (potential loss of signals in case of high data rates) due to insufficient specification, deadlock after loss of a signal.

In the following chapters the experience with model transformation, code generation and V&V on level of the automatically generated code is reported. The activities were executed in the context of the ACG (Automatic Code Generation) [4] and ASSERT (Automated proof-based System and Software Engineering of Real-Time systems) [5] projects.

Chapter 2 gives an overview on the applied model transformations. Chapter 3 provides details of the transformation from UML2 and chapter 4 from 3ADL. Finally, in Chapter 5 conclusions are drawn.

2 OVERVIEW ON THE APPROACHES

The models were established in the notations of UML2 and 3ADL and then transformed into the notation of ISGL. 3ADL (ASSERT AADL) is an extension of AADL defined by the ASSERT project. AADL [6] is a notation defined and maintained by SAE, the Society of Automotive Engineers.

AADL supports model-based embedded systems engineering. It is standardised and has been developed by and for the avionics, aerospace, automotve, and robotics communities. Textual and graphic notation with precise semantics for embedded and real-time systems, and UML profiles are supported.

ISGL is a language to model distributed real-time systems expressing their behaviour by Finite State Machines (FSM). Algorithmic code can be plugged in by functions executed during state transitions. Such functions are automatically integrated by the ISG tool, when generating the distributed execution environment.

The ISG tool supports verification of a model prior to code generation. Code is only generated when the static checks are passed successfully. To observe the properties of the code derived from the model "observers" are added to the generated code automatically. The observed properties are presented in textual and graphical form and in an rtf-document automatically generated after a run.

2.1 Transformation from UML2

The used model reflected at high level the functionality of a so-called Telecommand and Telemetry Manager (TMTCMgr) which receives commands from ground control and reports to ground on the success of command execution. The model consists of four processes for managing of command processing, queuing, verification and routing. It was established by a separate team in the course of the ACG project and built using the TAU tool [7] from Telelogic.

At the time the model transformation was scheduled in the ACG project, the "XMI"-files exported by the TAU tool where not usable. Element IDs were not unique or references where not properly specified, so that the structure of the model could not be reproduced from the exported file even in theory.

Therefore it was necessary to re-establish the model using a different tool. Eclipse UML2 was selected, also due to the fact that libraries for XMI input and output where directly provided. The model was manually re-established using the UML2 editor of Eclipse.

The saved model XMI-file is taken as input for generation of inputs to the UML2ISGL bridge from which the ISG generator produces the executables after successful verification of the model expressed in ISGL notation.

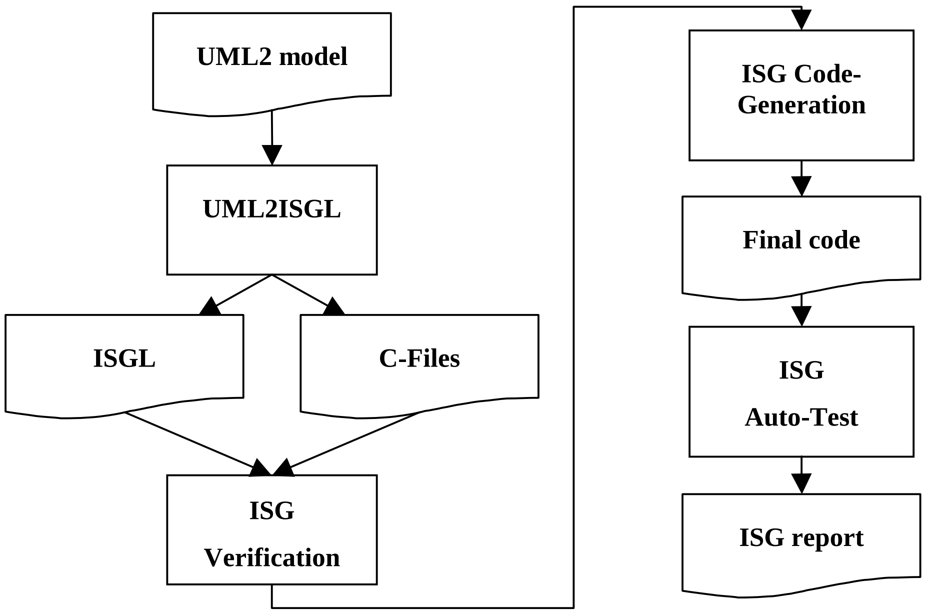
From the UML2ISGL bridge an ISGL file and C files are generated. The ISGL file defines the behaviour, performance and distribution of the processes. The C files include functions implementing the algorithms defined on UML modelling level. These functions are plugged into the FSMs at the corresponding state transitions.

The transformation only considers declared active classes (processes) and signal types. The FSM used for Use cases as well as message sequence charts (MSCs) are not considered and were not transposed to the Eclipse. They are not relevant for code generation and also do not provide any further formal refinement of the properties expressed in the model which could be used for verification or validation.

MSCs in particular represent the modeller's notion of how the internal system communication should look. In the specific case, the modelled MSCs were to be considered informal and took more of a documentation role than being an actual specification.

If at all, MSCs can be considered as test templates, which were to be overridden by the automatic and more comprehensive test cases automatically generated by ISG.

Fig. 2-1: Transformation from UML2 to ISG incl. Auto-Coding and Auto-Testing



2.2 Transformation from 3ADL

The TMTCMgr as established during the ACG project was re-built in 3ADL. It was restructured to specify architecture including system decomposition, distribution and communication channels.

In the course of the ASSERT project, an UML2 profile for architectural specification in analogy to AADL was defined. An experiment was exercised to extend the previously described UML2ISGL transformator to accept such 3ADL-based models to provide further tool diversification and analysis of the profile for applicability in terms of verification and code generation.

Both the 3ADL architectural profile and AADL mainly focus architectural modelling, so that at the time the experiment was executed no definitions on the specification of behaviour in 3ADL models were available. Therefore the concepts used in the previous UML2-to-ISGL-transformation experiment were reused.

However, while in the previous experiment architectural information such as CPU-mapping and communication channels had to be made up, the 3ADL profile shall provide a defined framework to actually model such information.

Unfortunately, at the time the experiment was executed, the 3ADL profile specification was incomplete regarding the specification of bindings of software components to hardware. Therefore this information could actually not be used.

However, as the concepts of ports and port connections was more precisely defined in 3ADL – based on its AADL roots – than in pure UML2, this feature was now properly used to describe the connections between the components.

3 THE TRANSFORMATION FROM UML2

3.1 Verification and Validation

3.1.1 Static Checks on the Transformed Code

By the static checking tools of ISG it was detected, that the UML model was formally incomplete. Some signals were identified which were either sent but not received within the model, or vice-versa. It had to be derived - by analysis - that these signals are intended to cross the system border. The entity receiving resp. sending these signals therefore was assumed to be the ground station.

While the topic of the model was clearly identified to be the telecommand manager only and therefore the model cannot be expected to cover all parts of the system, including the full on-board and ground system, the only formal indication on signals crossing the system border is a lack of sender or recipient within the model.

Without any explicit declaration of external interfaces in the formal part of the model (i.e. not including use cases), this inhibits any chance for detecting formal incompleteness of a model based on the sender-recipient relation. The respective signals could be marked as external signals or TMTC signals by an appropriate stereotype in the UML model.

3.1.2 Property Analysis of the Model and the Generated Code

The evaluation of the properties of the model and the generated code concentrates on the following feedback:

- coverage of states and state transitions

  As the model shall be complete and correct, every state and state transition shall be covered. The number of state transitions, especially the branching ratio, shall be as expected.

- timing diagrams

  The periodic occurrence of signals for all processes indicates that the FSMs can continuously be processed. When the periodicity is clearly visible, no overload occurs.

- Message Sequence Charts (MSC)

  The signal flow and the signal contents can be analysed by the information provided with the MSCs. By filtering of the provided information further properties can be evaluated like the number of lost signals.

Equivalent and even more information can be obtained from the textual report on properties, e.g. performance properties on bus and CPU utilisation[1]. However the analysis of the above information was sufficient to verify the model and identify the faults described below.

3.1.2.1Coverage

The coverage of states and state transitions is shown in Fig. 3-2.

Along each transition the associated incoming message of the state left is printed together with the line number in the ISGL/CPT file and the number of occurrences of this state transition. This latter information allows verification of the FSMs and of the generated code: the sum of state transitions by which a state is entered should be identical with the number of transitions by which it is left.

Moreover, it can be verified that the transition from the initial state to the first operational state only occurs once (see e.g. process "pktmanager", state "pseudo_start", incoming message "start_cmd"). Similarly, the number of state transitions for process "stimulator" into state "running1" and "running2" should be identical (possibly ± 1 depending on when the termination event occurred), because the stimulator toggles between both states.

Non-covered states and state transitions are indicated by red colour.

For two processes "pktmanager" and "queue" states and transitions are not covered.

In process "pktmanager" the "else"-branch of the "if" on the value of "PID" – the data item transported with the incoming telecommand – is not covered, with the other branches being covered for the values of "PID" considered to mark valid incoming commands. A deeper analysis yielded that the "else"-branch is dead code (in the original model), as for an invalid command a previous branch would forbid reaching the given "if" in any case.

In process "queue" a state and two state transitions are not covered. Analysis of the (original) model shows that the state and state transitions are related to loss of a signal. The queue process keeps a counter that represents the difference in number between the number of validation requests sent to the "verify" process and the number of answers received. Whenever this counter is non-zero on receiving a new telecommand it can be deduced that some of the validation requests or answers were lost in transit.

[1] Such performance properties were not part of the original model, and no requirements are known. Therefore verification of such properties is out of scope.

Therefore a new generation cycle is executed, enforcing loss of signals at a probability of p=0.1. The results are shown in Fig. 3-3.

Having identified the uncovered state related to the loss of signals in process "queue" a fault in the fault-tolerant algorithm was detected which should protect against loss of signals.

3.1.2.2Analysis of the Control Flow

The behaviour, the control flow and performance constraints were analysed by timing diagrams and message sequence charts (MSC) for the nominal and non-nominal (fault injection) cases. Subject of fault injection was loss of signals.

Loss of signals was considered for the following cases:

- only one process can lose signals,
- all processes can lose signals (pktmanager, queue, routing, verify).

Surprisingly, the state and the two transitions of "queue" are still not covered, and even states and transitions of pktmanager and routing are not covered. The explanation is: a deadlock occurred.

To explain the origin of this deadlock it is necessary to have a more detailed look at the control flow (c.f. Fig. 3-1)

As already explained the process "queue" maintains a counter representing the difference between the number of validation requests sent and validation answers received. This counter is checked whenever a new telecommand arrives at the process "queue" for queueing.

The source of the telecommand arriving at "queue" is the process "pktmanager". However, after receiving a telecommand, "pktmanager" enters a different state where further telecommands from ground will be ignored and therefore cannot be forwarded to the process "queue".

The process "pktmanager" only leaves this state for the state in which telecommands can be processed when the result of the verification is received. If, however, either the message containing the verification result or even the request for verification as sent by "queue" is lost, the process "pktmanager" will not return to this state.

The process "queue" will therefore never get the telecommand which is needed to actually trigger the signal-loss-detection code. The system deadlocks.

3.1.2.3Some Remarks on V&V of the Model

Regarding the loss of TCs this could be a matter of on-ground processing as well, but there is no evidence in the received UML model, where such problems shall be solved. Therefore from a rigorous verification point of view it is stated that the TMTCMgr does not tolerate loss of TCs or loss of internal signals in the shape defined in the received UML model.

A synopsis of the faults found in the model during the transformation exercise is presented in Fig. 3-4.

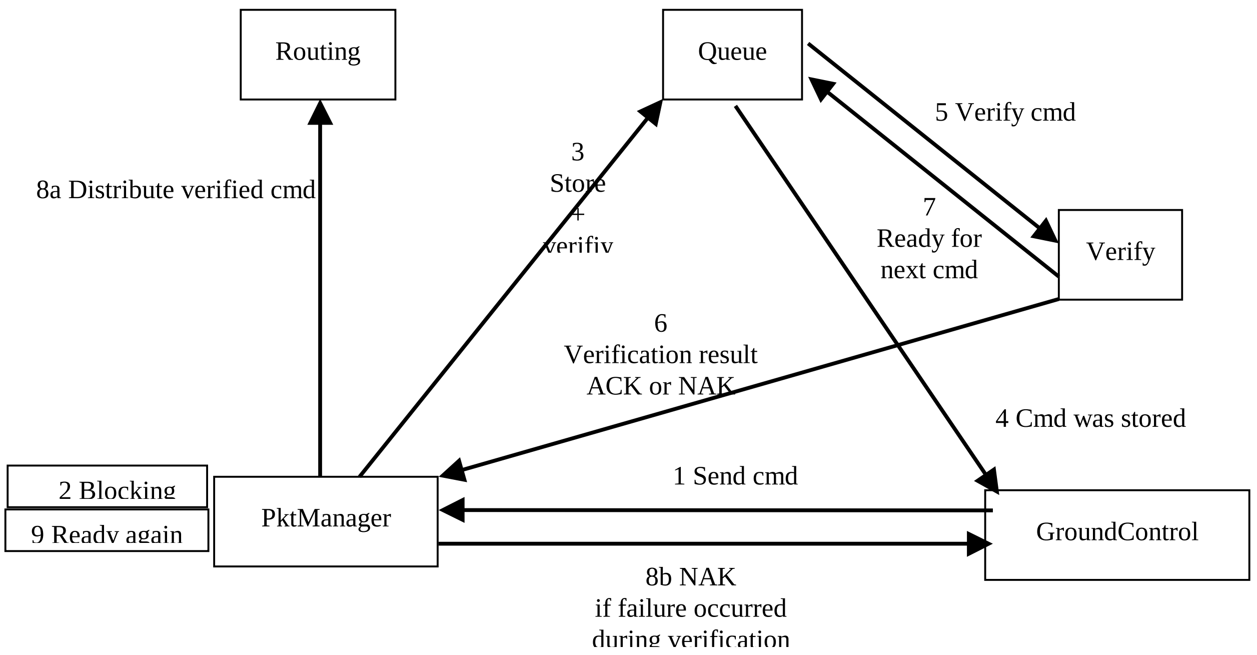
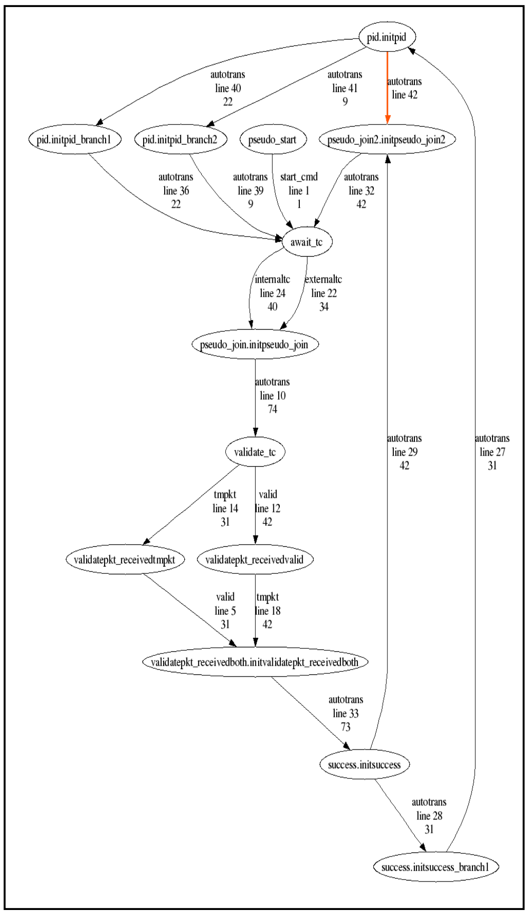


Fig. 3-1: Control Flow of the Command Manager


pid.initpid
pid.initpid_branch1
pid.initpid_branch2
pseudo_start
pseudo_join2.initpseudo_join2
await_tc
pseudo_join.initpseudo_join
validate_tc
validatepkt_receivedtmpkt
validatepkt_receivedvalid
validatepkt_receivedboth.initvalidatepkt_receivedboth
success.initsuccess
success.initsuccess_branch1

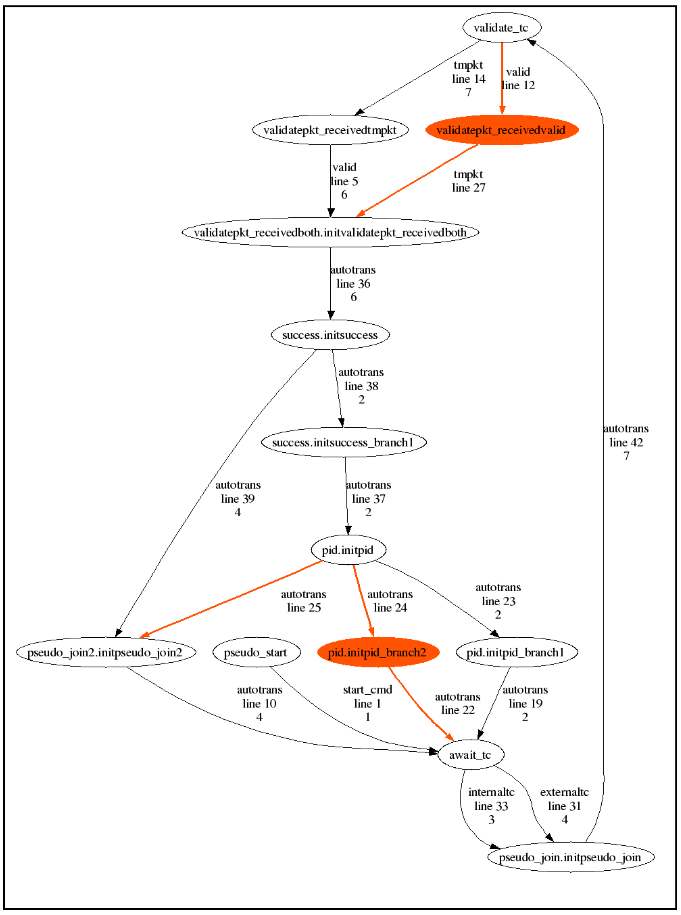

validate_tc
validatepkt_receivedtmpkt
validatepkt_receivedvalid
validatepkt_receivedboth.initvalidatepkt_receivedboth
success.initsuccess
success.initsuccess_branch1
pid.initpid
pseudo_join2.initpseudo_join2
pseudo_start
pid.initpid_branch2
pid.initpid_branch1
await_tc
pseudo_join.initpseudo_join

Fig. 3-2 : State Transitions without Loss of Signals

Fig. 3-3: State Transitions with Loss of Signals

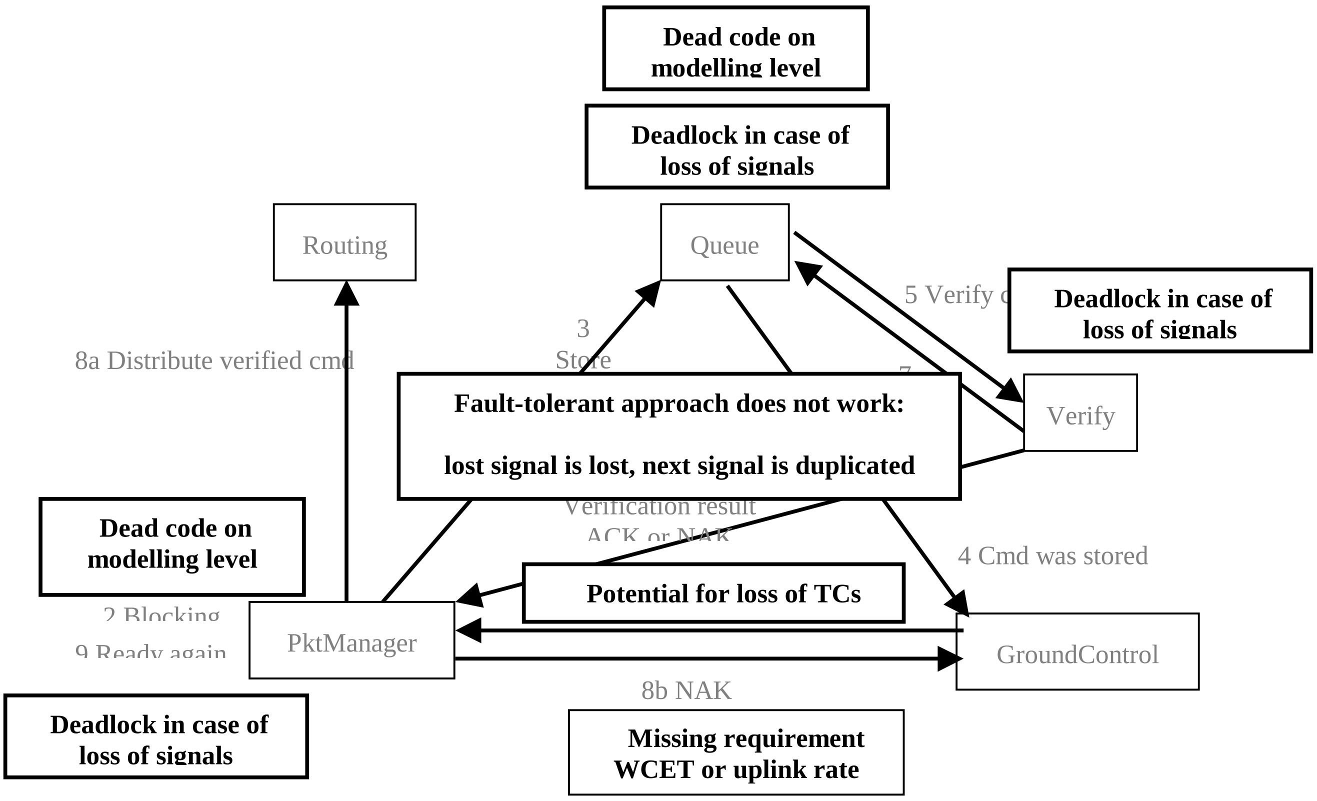

Dead code on modelling level
Deadlock in case of loss of signals
Routing
Queue
5 Verify c
Deadlock in case of loss of signals
3 Store
8a Distribute verified cmd
Fault-tolerant approach does not work: lost signal is lost, next signal is duplicated
Verify
Dead code on modelling level
Verification result ACK or NAK
4 Cmd was stored
Potential for loss of TCs
2 Blocking
9 Ready again
PktManager
GroundControl
8b NAK
Deadlock in case of loss of signals
Missing requirement WCET or uplink rate

Fig. 3-4: Faults found in the model by automatic test

3.2 Conclusions on Model Transformation

3.2.1 Principal Considerations

The Unified Modelling Language (UML) is actually a set of languages due to the presence of semantic variability points (SVPs).

When transforming an UML2 model to another notation like ISGL with the intention of code generation, all such ambiguities of the UML2 modelling language have to be resolved, analogous to the descent in abstraction levels normally performed on manual transformation of an abstract model into code (Fig. 4-1).

While in manual coding a developer would use experience and additional knowledge of information outside the model to concretise the definitions of the model, automatic code generation, (equivalent) transformation or verification require the explication of such information.

The information required for concretisation in the automatic case must be present either in the model itself or must be automatically deductable from it by the semantic rules of the language. In order to reduce complexity and thereby errorproneness and effort for the transformation, the deduction should be straightforward.

There is a third place in which the information may be present: The transformator or code generator. However, in this case the information is essentially hidden from users of the modelling language and the transformation. Formally, the transformator does constrain or modify the semantics of the language and thereby actually does not comply with the specification.

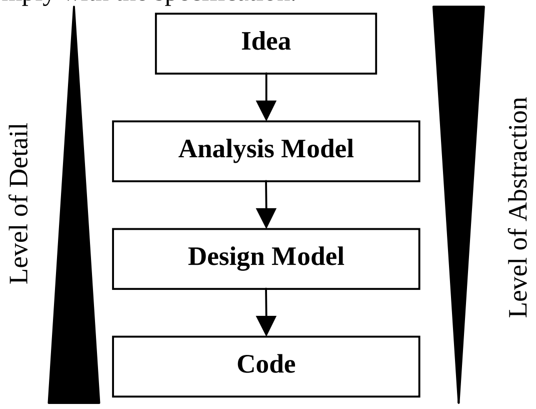


Fig. 3-5: Detail and Abstraction in Models

In cases where the required information is not present in either the model or the semantics of the language, either of both must be extended, e.g., explicitly by introducing additional information when developing a model transformator, as was done in this experiment.

It also has to be considered that incomplete or missing explicit semantics make verification of such a model meaningless, as missing semantics imply missing constraints for the modeller regarding "building the system right".

In case of the provided TAU model, no performance issues were modelled, as UML2 does not require the specification of performance figures, even it does not support such a feature at all for verification purposes.

While it could be argued that this is essentially a flaw of the model, it is also an indication that UML as a language is not well suited for modelling realtime systems.

The modelling language is also to be considered as part of the tooling approach and therefore has to fulfill the same requirements as the tools themselves. Although there are many off-the-shelf languages and tooling solutions available on the market, a principal "make-or-buy"-decision based on a rigorous analysis process is to be made for a language and the associated tools like for any other software-based system or part thereof.

Tools are typically deployed to simplify the work of their users so that these can concentrate on the relevant parts of their work. Productivity cannot be considered as rising if the products are qualitatively inappropriate.

Therefore tools must also guide their users to avoid mis- and underspecification. This clearly is not the case with most UML-tools, which are designed for general-purpose activities due to marketing considerations and therefore cannot consider specific needs in specific projects. Still the transformation of the abstract UML2 model into another abstract modelling language (ISGL) together with the setup of a mockup for automated stimulation of the system in ISGL as required by this modelling environment identified the performance issues.

3.2.2 Specific Choices

The structure-based approach applied in the UML2 model therefore had to be transformed into the specialised structural template of ISG. It was therefore assumed that each class would contain at most one Finite State Machine. All classes containing such a state machine were considered to represent ISG process types with exactly one process instance. All other classes were ignored, as in this experiment, such classes were only declared but not used in any part of the model. The presence of such unused classes is an evidence of further dead code in the model, at least in its current shape. Currently, no message is issued on detection of such dead code by the UML2ISGL bridge, but this is a future issue for the transformator raised by this discussion. Whether such classes would have been used in a future version of the model is not known.

Relationships between classes were also ignored, as ISG does not require or allow the specification of particular relationships between processes or process instances in a similar manner. Instead, such relationships are expressed by the actual presence of interaction between the processes or process instances in the behavioural part of ISG.

No information regarding the distribution of the processes among available CPUs was provided in the UML2 model, therefore all processes were assumed to be located on the same, single CPU. Furthermore, no information about the channels between processes was provided in the UML2 model, therefore a general, single virtual bus was established.

UML Properties declaring attributes of a class or a Finite State Machine were transformed directly into process instance variables in the generated C-files.

The signal definitions available from the declared interfaces were used to establish the order and format of ISG signal attributes in the packets sent between processes.

With the introduction of a counter-piece to UML2 pseudo-states, translation of UML2 Finite State Machines into ISG Finite State Machines is almost straight-forward. However, four issues of ambiguities either in semantics or in the relationship between concrete and abstract UML2 syntax had to be resolved.

3.2.2.1Accessing Signal Attributes

The first issue concerned access to attributes of received signals. While in diagram form the target variables for values from the signal are written in parenthesis behind the name of the triggering signal type, thereby declaring the way to access these attributes, this construct cannot be directly translated into the abstract syntax. The UML2 standard defines that attributes of signals in expressions may be accessed as <signal-name>.<attribute>. However, no clear definition is presented on how to retrieve the object representing the signal for use in a ReadStructuralFeatureAction. It was therefore defined that a ValuePin with expression "signal" shall denote the object instance representing the recently received signal.

3.2.2.2Transitions triggered by multiple Signals

Secondly, in process "PktManager" one transition from state "ValidatePkt" is triggered by the reception of two signals. Due to the fact that the attributes of both signals were used in the effect activity of the respective transition it was to be assumed that both signals had to be received in order for the transition to fire. As the retrieval and storage of signal attributes had to be explicitly expressed by ReadStructuralFeatureAction in the abstract syntax, the transition was separated so that the two signals were received one after the other, triggering an additional transition in between. As it could not be found from the model whether the signals would always come in a specific order – evidence of the opposite was found – two alternatives were introduced to handle both possible orders.

The second alternative was added manually on UML2 level. In principle, this could also be handled by the transformator given that the addressed signal would have been uniquely identifiable in the ValuePin of the ReadStructuralFeatureAction elements accessing signal attributes. In that case, all trigger signal types would have to be associated with the respective transition and actions would have to be provided – in this case in the process of transposing the model from diagrams to the abstract syntax – extracting the attributes of each signal into the specified process instance variables.

The transformator would then have to extract the parts relating to the access to each of the signals and introduce additional alternatives for each possible order of reception, associating the extracted access actions for each signal to the appropriate transitions.

The implementation of this process was considered too time-consuming regarding the schedule of the project and in view of only one such case in the model.

In general, the semantics of multiple triggers on a transition in UML2 is undefined. Multiple triggers on a single transition can be interpreted as expressing that the transition is triggered by

- the reception of either signal, or
- the reception of the signals in the given order, or
- the reception of the signals in any order, or
- the simultaneous reception of the signals – in case of synchronous communication with "zero-time" semantics.

It was only possible to associate one specific of these semantics to the multiple triggers in this model as

- the attributes of the two signals were assigned to different process instance variables and both variables were used in the transition effect, implying the requirement that both variables must be initialised before the transition could be triggered, and
- the fact that the two signals were sent in both possible orders from different origins in the model.

This means that the actual semantics were defined by the context of the model and not by the modelling language or an a-priori specification of the semantics used.

This example shows the conflict between the intentions of the UML2 authors and the goal of code generation.

3.2.2.3Generic Actions and Conditions

It was a goal of the UML2ISG bridge design to use as many explicitly available elements of UML2 as possible to directly express the diagrams. While opaque elements such as Activity or Note are available, from a transformator point of view these essentially hide information or require the introduction of possibly sophisticated parsers for the contained textual information. This is not always possible, either due to lack of human resources, schedule and budget or due to lack of any formal grammar which would apply to that user-defined information.

Almost all elements of the diagrams could be expressed directly, except for guard expressions on conditional transitions from choice pseudo-states and for one increment and one decrement expression in process "Queue".

Therefore it was defined that the transformator would expect expressions and text inside opaque Activity nodes to represent valid C-code. This way, these expressions and code fragments could be directly inserted into the generated C-UDFs. The requirement was already met in all actions which were to be expressed by opaque Activity nodes. However, most of the conditions present in the model had to be rewritten during the manual transformation to abstract syntax.

In addition it was defined that any expression of the form "self.<var>" represents a reference to the process instance variable with name "<var>". All expressions and opaque Activity nodes had to be rewritten in this way during the manual transformation to abstract syntax. The transformator replaces these expressions by the appropriate code as required to access the given properties in the context of the generated code.

3.2.2.4Missing Target Process Specifications

Also, no target was specified for the send-signal-actions in the Finite State Machines anywhere in the UML2 model. As port specifications on the classes seemed to imply that any signal type is handled by at most one class/process, the specification of the destination process required by ISG was found by searching for a process handling the given signal.

This worked in all but one case, where the outgoing signal "inqueue" crossed the system border of the model. A special rule was established by which all such signals were to be sent to the automatically generated process "initproc" (which controls system initialisation), which would merely receive the signal but not react on it.

3.2.3 Summary Conclusions

A considerable amount of issues regarding code generation from UML2 models was identified and resolved.

The experiment shows that most of the semantical ambiguities were already resolved in the transformation from concrete to abstract syntax, an activity which was not planned at all. This leads to the conclusion that the UML2 abstract syntax fits better with the constraints of code generation, although this is only valid in the context of the applied elements of UML2 and the specific model used.

Also modellers must take care not to make extensive use of the many opaque elements of UML2. These are intended to allow general applicability of the modelling language, a goal which in fact contradicts the goal of strict verification.

In addition, the issues regarding the transformation from the structure-based concept of UML2 to the behaviour-based concept of ISG shows that the definition of a UML2 profile is constrained by the basic structure of UML2.

Furthermore, information may be present in the model which cannot be found or processed by a transformator. The resolution of the ambiguity regarding multiple signals as transition trigger provides an example for this.

Resolution of such semantic ambiguities may not only be important to allow code generation at all. It is, of course, another goal of code generation to generate efficient code. If a transformator or code generator had to fall back to the most general resolution in some cases, this would most probably also be the most inefficient solution.

While the authors of the UML2 intended that the semantic variability points be resolved by each team to create a tailored instance of the UML to be used, practice tells us that this intention is not followed by the users of the UML2.

Part of the reason for this may be that the number of known semantic variability points – i.e. those that have been included in the UML2 on purpose – is overwhelming and a complete resolution seems impracticable to many UML users.

In addition many points have been identified by researchers where the semantics of the UML2 are incomplete but without any indication that this incompleteness was intended as such by the authors of the specification. These points can be seen as unintended semantic variability points.

It is difficult not to reason whether a language so complex that even its creators have difficulties defining a sound semantic is a good choice for everyday use by non-language-experts in the field of critical systems where a sound semantic is absolutely crucial for the success of a project.

This raises the question whether to follow the UML2 track and apply profiles, requiring

- additional verification means to identify conflicts, which then have to be manually solved,
- additional and possibly complex means for transformation to the application domain,

or to define domain specific languages which are directly targeted at the verification and code generation requirements driven by the application domain, e.g., distributed real-time applications. Where domain-specific languages only allow such constructs which can be successfully transformed into efficient and correct code, UML2 and profiles may allow constructs which will cause problems on automatic transformation and implementation.

4 THE TRANSFORMATION FROM 3ADL

4.1 Verification and Validation

Regarding behaviour and performance the same V&V measures were applied as in the ACG experiment.

However, the specific concept of hierarchical decomposition and the resulting need for delegation of communication between hierarchy levels lead to additional points for verification.

As a connection between two ports can consist of a number of partial delegate and sibling connections at different component levels and at each level a bus binding for that part connection may be specified, both 3ADL and AADL allow to specify port connections which are bound to different busses at different hops. This would mean that the ports which are shared by the part connections bound to different busses would essentially take the part of a transceiver.

Even if this would be considered legal at all in AADL, it should be considered bad practice. After all a port under most circumstances is merely a modelling placeholder required to facilitate hierarchical decomposition and connection delegation. It is not an active component which can take the part of a transceiver, in contrast to, e.g. the device AADL component.

This example shows that unfortunate selection of concepts for modelling languages can introduce additional verification problems, although this specific case it is probably tolerable because a check for such a condition can be implemented without much effort, informing the user about such an inconsistency.

4.2 Conclusions on Model Transformation

4.2.1 Principal Considerations

The 3ADL concept as available at the time of definition of the required mapping concentrates on architectural modelling. The definition of behaviour is essentially undefined or at most implied by the use of AADL standard properties for specification of source code for threads and subprograms.

For 3ADL a complete mapping of AADL to UML is not yet established. Similarly, the 3ADL to ISGL mapping does not support all possible 3ADL constructs. For example it was decided that threads behaviour may only be specified in the form of a UML state machine defined in a UML class representing the thread type.

The mapping from 3ADL to ISGL is defined by several steps. In the first step the instantiated structure of the specified system is derived. Connections for event, event data and data ports are resolved.

In the second step a ISG process type with a single instance is defined for each 3ADL thread instance. This corresponds to a flattening of the hierarchical structure of the 3ADL system definition.

The FSM for the ISG process type is generated from the UML state machine associated with the UML class defining the 3ADL thread type. This association is outside the scope of 3ADL.

4.2.2 Specific Choices

In contrast to the ACG experiment, no elements from the model were actually ignored, as it was defined according to the concepts and modelling guidelines of the 3ADL profile. The language elements used are therefore more suitable for the use in specification of distributed real-time systems and matches the concepts of ISGL more closely.

5 CONCLUSIONS

In comparison of the two modelling approaches by the performed work the 3ADL approach is clearly more suitable for the modelling of distributed real-time systems than generic application of UML2, as it more strictly defines both the domain of allowed modelling elements, but also the modelling process.

This leads to the conclusion that a constructive approach to the definition of domain specific languages, i.e. building up the set of modelling elements, is more reasonable than a destructive approach, i.e. reducing the set of modelling elements from a generic, already existing language.

The results presented above also show that model transformation can work, but imprecise modelling notations may prevent a reasonable transformation. In the investigated example certain assumptions could be made to make an automated transformation feasible. However, such a transformation might not be reusable for any other model. A manual transformation should not be considered as it is inefficient and may introduce errors.

On the other side the usefulness of model transformation was demonstrated regarding complementing tool capabilities for verification and validation, especially for identification of non-anticipated faults.

Surprisingly, “non-anticipated” got a meaning not considered before: presence of faults not detected by a tool is to be understood as “non-anticipated”, as the tool suggests that no faults exist.

However, as a practical experiment with different modelling notations it also shows that proper requirements engineering w.r.t. the intended use – application domain, verification, code generation, tooling in general – is absolutely required.

It is not enough to establish some language based on previous manual design practice. Meta-modelling cannot work miracles based on the old principles. Information in a model needs to be presented in a way which provides both ease of use for the modeller and is easily and directly accessible to tools.

Essentially, building blocks should not only manifest themselves in the form of reusable model parts. They should also be explicitly be considered when defining the meta-model. Instead of copying a detailed model for the same component from project to project and manually adopting the model to the interface needs of the new project, the component should simply be declared as present and tools should take the part of actually modelling it. ISG, for example, allows definition of telecommand handling simply by declaring the existence of telecommand literals. No explicit modelling is necessary.

Finally, the observed results show that more effort needs to be put on precise semantics of modelling notations in order to allow a higher degree of diversification and model exchange based on automated model transformation.

6 REFERENCES

[1] Object Management Group (OMG), “UML 2.0 Superstructure Specification: Revised Final Adopted Specification”, Version 2.0,, ptc/04-10-02, October 2004.

[2] 3ADL/ASSERT AADL: Patrice Boisieau, Nicolas Gianiel, Reconciling the Needs of Architectural Description with UML, Deliverable D4.1-3 Issue 1 Revision 1, April 2006

[3] ISGL, ISG Language, http://www.bsse.biz/products/isg

[4] Automatic Code Generation (ACG), ESTEC contract no.18670/05/NL/GLC, Noordwijk, The Netherlands, 2004-2006, ACG-TN-6-SE-BSSE, V1.2, 16.11.2006, Report on Experiment #3: The Use of ISG, DARTT and DCRTT,

[5] ASSERT: Automated, proof-based System and Software Engineering of Real-Time systems, Integrated Project 004033 of FP6 of the European Union, 2004-2007

[6] AADL, Architecture Analysis and Design Language, http://www.aadl.info

[7] TAU tool from Telelogic, http://www.telelogic.com/products/tau/index.cfm