Automated Verification of Code Generated from Models: Comparing Specifications with Observations

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 interest for automatic code generation from models is increasing. A specification is expressed as model and verification and validation is performed in the application domain. Once the model is formally correct and complete, code can be generated automatically. The general belief is that this code should be correct as well. However, this might be not true: Many parameters impact the generation of code and its correctness: it depends on conditions changing from application to application, the properties of the code depend on the environment where it is executed.

From the principles of ISVV (Independent Software Verification and Validation) it even must be doubted that the automatically generated code is correct. Therefore an additional activity is required proving the correctness of the whole chain from modelling level down to execution on the target platform.

Certification of a code generator is the state-of-the-art approach dealing with such risks,. Scade [1] was the first code generator certified according to DO178B. The certification costs are a significant disadvantage of this certification approach. All codes needs to be analysed manually, and this procedure has to be repeated for re-certification after each maintenance step.

But certification does not guarantee at all that the generated code does comply with the model. Certification is based on compliance of the code of the code generator with given standards. Such compliance never can guarantee correctness of the whole chain through transformation down to the environment for execution, though the belief is that certification implies well-formed code at a reduced fault rate.

The approach presented here goes a direction different from manual certification.. It is guided by the idea of automated proof: each time code is generated from a model the properties of the code when being executed in its environment are compared with the properties specified in the model. This allows to conclude on the correctness of the whole chain for every application and related generated code.

1 INTRODUCTION

Faults may be introduced in the chain from modelling to target execution either by a model itself or by the code generator.

If a model is the source of a fault, modification is actually part of the normal development cycle of an application and does not pose a problem, provided its identification is supported by a code generator e.g. by comprehensive presentation of the properties as observed in the execution environment..

If the source is the code generator, the needed modification of the generator will lead to loss of its certification. The alternative, modification of the generated code, will lead to loss of previous verification and certification of the model and the generated code.

Manual analysis of properties of the executed code for evaluation of its correctness is also very expensive, and also not free of potential human errors. However, an automaton cannot replace an engineer for this task, because oracles are not available to predict what is correct or not (this is at least true in most cases, only in simple cases an oracle may exist).

An automated proof would be rather helpful to conclude on the correctness of large amounts of code as usually produced by code generators. This issue leads to the question how an automaton can contribute to verification and validation though not knowing the results, thereby significantly reducing costs and time.

As an automaton does not know itself what is correct or not in an application, a practical solution must rely on a proof for which the automaton does not need to evaluate the contents of application-specific properties.

This conclusion guides to a simple comparison of two sets of properties: if both sets comply, the result is correct. The principal problem is that the model set ("specification") and the set of properties ("results")

observed when executing the generated code cannot be taken for such a comparison, because they are not equivalent from the automaton's point of view.

Apparently, the observed properties need to be transformed back into a model representation to get two equivalent representations. This is the approach chosen.

This paper will present and discuss this approach and first results obtained rom practice in the context of the ISG modelling environment [3]. In chapter 2 the principal approach to verification of the whole chain is explained. Chapter 3 identifies the steps needed in practice for auto-proving. Chapter 4 discusses the current results. In chapter 5 the chosen approach is compared to model verification based on Message Sequence Charts (MSC) [4]. Finally, conclusions are drawn in chapter 6.

2 THE VERIFICATION CHAIN

The ISGL language [3] is centered around behaviour, performance (timing and scheduling), communication (by signals) and distribution of processes across a platform. Therefore these properties will be subject of observation (which is already inherently supported by ISG) and back-transformation into the reference domain of a model.

The advantages are:

- cost reduction due to automation of the verification process for the code generator,
- immediate proof of correctness for every application and each of its versions based on observations in its real environment.

Fig. 2-1 shows the principal verification chain as applied by the verifier of the ISG code generator (CGverifier):

- automated transformation from the reference model domain into distributed, instrumented executable code,
- automated observation of the relevant properties when executing the generated code in its intended environment,
- automated transformation of the observed properties back into the observed model,
- automated comparison of the reference and the observed model.

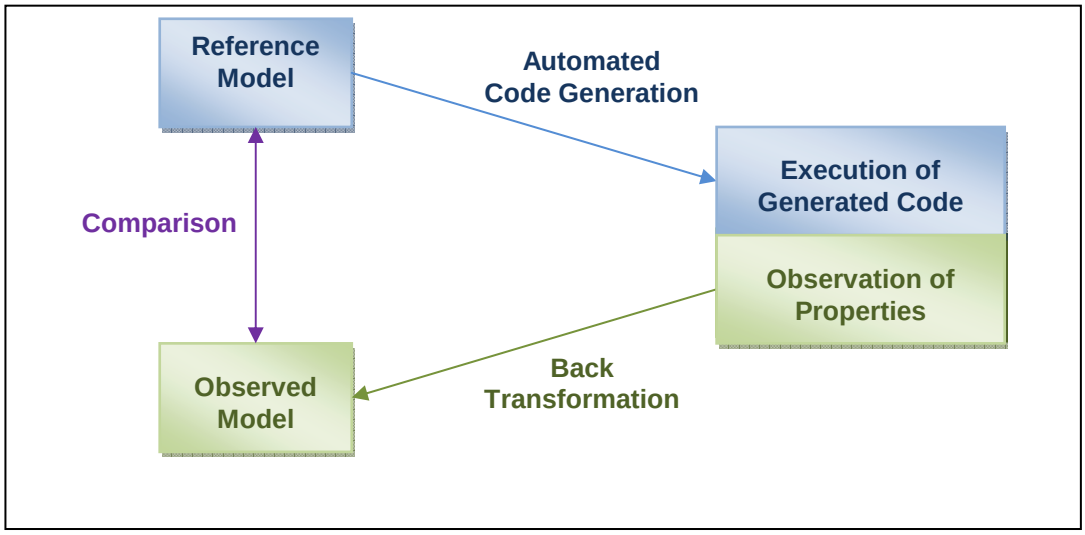

Fig. 2-1: The Automated Verification Chain

From the compliance of reference and observed model the correctness of the generated code can be derived, provided that

1. the code generator and the back transformator are independent,
2. a fault in one of both transformators is not compensated by a fault in the other transformator.

Condition 1 shall be satisfied by independent implementations which only interface with minimum information on the relationship between properties and modelling elements. In fact, the back transformation requires a quite different and probably simpler implementation than code generation, which makes it easier to assess its correctness.

Condition 2 cannot be formally satisfied, but the probability should be rather small due to the independency of both generators, especially compared to the fault probability human beings may introduce into a certification process. A cleanroom approach can be applied to the development of forward and backward generators, but the fact that different, though

communicating teams implement the code generator resp. the verification facility should provide acceptable credibility and reliability of the results.

3 THE VERIFICATION STEPS

3.1 Observation of Properties

The properties of the code shall be observed from messages issued by the "application" (the generated code) when

- application signals are exchanged between the processes,
  including information on source and destination process and their instances, the current state and the message type,
- timer signals are issued
  providing information on scheduling and timeout events
- additional information signals are issued,
  providing information e.g. on state transitions and timing requests.

The observed sequence of such signals does allow to construct a model equivalent to the reference model w.r.t to the properties to be verified. This implies that in a first step of the CGverifier only certain modelling elements are considered for the comparison, the ones which can be derived from the supported observations (see Table 3-3).

An essential point of the chosen approach is that

- only one incoming signal is processed at any instant of time, and
- no, one or a sequence of signals is issued during processing of such an incoming signal,
- messages on atomic activities and the current and next state are issued.

Branches in such processing steps due to conditions can be identified by the number of occurrences of signals following an incoming signal.

3.2 Back Transformation and Comparison

From the observed sequence the back transformation will construct the Observed Model which should look like the equivalent of the model or parts of it (see 4.3 for a discussion on this point), so that the comparison should yield an "OK", or "not OK" in case of non-compliances. For this transformation only the intended operational semantics of the modeling language together with the observation interface need to be known.

The transformator will track the states and incoming messages and correlate any following out-going message, while evaluating different streams from process instances to derive a process type as envelop of such instances.

The Observed Model will only represent such code parts which have been executed.. E.g. if a timeout is not observed, the observed model may lack the related part. Consequently, this implies that a successful comparison can only be reached when during observation full statement and decision coverage (nodes and edges of the code structure) of the reference model is achieved. In case faults are handled in the model, the faults should occur, which may imply the need for fault injection during execution.

The need for full coverage may increase the test effort, however it also ensures a well tested model and implementation.

Identification of performance properties mostly depend on statistical observations of quantities like jitter of periods, timeouts and data lengths. Therefore an exact match cannot be required, and verification is limited to the compliance of the observed sub-range with the reference range.

3.2.1 Monitored Items

Verification is based on the items listed in Table 3-6. They are grouped into

- state items
- distribution items
- items of messages
- items related to timer operations
- items confirming execution of a UDF
- general items.

For each item the source is given from where the information is provided:

- model ("model")
- generated code ("code")

and for which purpose the information may be used:

- information only ("info")
- cross-checks ("cc")
- verification of the code generator ("vv")

For verification of the code generator (vv) only minimum information from the generated code is allowed, originating at the basic locations in the generated source code and the underlying run-time system provided by ISG. However, additional information may be useful

- to check the contents of the *Observed Model* internally in the cgVerifier by information declared as "info", and

- to apply cross-checks to the back transformation based on redundant information declared as "cc" to raise the chance for detection of a fault in the forward and backward chain.

Such items may contain model information or information from the generated code, depending on the location.

3.2.2 Verification Issues

Table 3-3 defines the verification issues to which a location contributes. This allows the construction of the Observed Model from the *observations*.

At each location in the code which is relevant for observation of a desired property, a message is issued.

In case of UDF calls (call of User-Defined Functions during a transition in a Finite State Machine) four levels are introduced to verify the call of a UDF up to the degree supported by ISG:

- a stub generated by ISG is executed as UDF,
- a call of an external UDF from a stub into which the call was inserted manually,
- a call of an external UDF from a stub and an interface function generated by auto-integration
- the direct call of an external UDF.

From the sequence of recorded properties a model is constructed by translating each such record into a statement of the ISG modeling language, yielding the “observed model” at the end.

3.3 Observation Sequences

The following principal sequences of signals shall be observed by which the properties of the generated code can be identified when it is executed.

3.3.1 Basic Sequences

Sequence of Activities

This sequence (*Table 3-1*) is the simplest one. It only consists of activities following an incoming message, and the state is not changed at the end.

Simple Transition Sequence

This sequence stands for the basic activities which can be observed in response to an incoming message immediately followed by a transition into a permanent state.

Timer Request

This sequence is observed when scheduling is required or a timeout condition is raised.

Sequence Type	Step
Activities	in-message
	activity
Transition	*Sequence of activities*
	state transition
Timer request	timer request sent
	timer request received

Table 3-1: Basic Sequences

3.3.2 Composed Transition Sequences

A number of compositions of simple and further composed sequences are possible (Table 3-2).

Multiple Transition Sequence

Such a sequence repeats the *Simple Transition Sequence* for intermediate transitions to (UML) pseudo-states, terminated by a *Simple Transition Sequence* entering a permanent state.

Entry/Exit Extended Sequence

This sequence adds activities before and/or after a transition into another permanent state. If both, the number of entry and exit activities is 0, it is equivalent to the *Multiple Transition Sequenc*e.

Timer Extended Sequence

In case a timer event is received, either for scheduling or handling of a timeout, it is converted into an equivalent incoming message, which is processed like any other incoming message, e.g. sent by another process. Table 3-2 only shows the *Entry/Exit Transition Sequence*, but a *Multiple Transition Sequence* is possible in this context, too.

Sequence Type	Step
Multiple transition sequence	*Simple transition sequence* to a (UML) pseudo-state
	Simple transition sequence to a permanent state
Exit / Exit Transition Sequence	*Simple transition sequence* to a (UML) pseudo-state
	Sequence of Activities
	Exit Activity *Sequence of Activities*
	state transition
	Entry Activity *Sequence of Activities*
Timer extended sequence	timer response event
	Exit/Enter Transition Sequence

Table 3-2: Composed Sequences

Property Group	Property Item	Verification Issue
Behaviour	Process State In-message State transition	The tuple *(process,state, inMsg)* must exactly match. All activities following an incoming message must exactly match. An activity is characterised by the tuple *(UDF call, outMsg, destination process, destination instance).* A sequence of activities begins with an incoming message and terminates when the next incoming message for the same process instance occurs. Such a sequence may include a state transition to a non-pseudo-state. The destination state may be the same as the current state. A terminating state transition always enters a non-pseudo state (in UML sense). State transitions occurring within such a sequence may enter a pseudo-state. A sequence must not terminate in a pseudo-state. A conditional state transition may occur at the end of a sequence. When a state is left, *onExit* activities may occur, i.e. a sequence may be extended by additional activities when a state is left. When a state is entered, *onEntry* activities may occur, i.e. a sequence may be preceded by additional activities when a state is entered. A sequence of activities related to *anystate* may be observed in every state. An activity related to *asyncstate* may be observed in every state.
Link to Code Extensions	UDF Call	The execution of a UDF must be verified on the lowest level possible: •in the generated stub •in the generated code integrating external C and Ada code •in the provided external code if observation is supported
Communication	In-Message	The received message and the receiving process instance must comply with the destination list included in the message.
	Receiver Instance	The destination instance must comply with the (max.) number of instances. All instances must have received one message, at least.
	Destination Instance	The destination instance must comply with the (max.) number of instances.
	Channel	The observed communication channel must comply with the reference channel.
Distribution	Process Instance	Every instance is mapped onto one logical processor only.
	Logical Processor	A logical processor is not mapped onto more than one physical processor..
Performance	Period	The observed range of a period must comply with the reference range. The observed timer modes must comply with the reference modes.
	Timeout	The observed timeout delay must comply with the reference range.
	Message Length	The observed timeout delay must comply with the reference range.

Table 3-3: Verification Issues

3.4 An Example

Table 3-4 shows the definition of a model in ISGL language [3]. The model consists of process types *sps* and *controller*. A process type is a template from which processes (instances of a process type) are derived by specifiying the maximum number of instances.

Each process is organised as a Finite State Machine (FSM). In every state a message shall be received, upon which a sequence of actions may be executed, terminated by a state transition into the same or another state. Pseudo-states may be entered, too (not used in this simple example). A pseudo-state is a state which may be entered from another state, only, but it is not entered on reception of an incoming message. When another permanent state is entered (not a pseudo-state) an exit and an entry action sequence may be executed to terminate the previous state correctly and to perform some initialisation of the next state (this feature is not shown in the example).

```
sps states:
   in state startup:
   on message poweron: send message startup to controller instance 1
      enter state identify
   end
   end
   in state identify:
   on message GetIdentity: reset timeout GetIdentity
   create timer motion with constant period 100 msecs ... 2 secs
   enter state init
   end
   on timeout GetIdentity:
   send message startup to controller instance 1
   expect reply GetIdentity within 5 secs
   keep samestate
   end
   end
   in state init:
   on timer motion:
   send message ReadyToLoad to controller instance 1
   enter state idle_WaitExec
   end
   end
controller states:
   in anystate:
   on message startup:
   send message GetIdentity to process sps instance 1
   keep samestate
   end
   end
   end
```

Table 3-4: An Example of an ISGL model

Active Process		**Message State**		**Message /**	**Destination Process**	
Process	Instance	**Type**		**Parameters**	Process	Instance
sps,	1,	in,	startup,	poweron		
sps,	1,	out,	startup,	startup,	controller,	1
sps,	1,	trans,	identify			
controller,	1,	in,	startup,	startup		
controller,	1,	out,	startup,	getidentity,	sps,	1
sps,	1,	in,	identify,	getidentity		
sps,	1,	period,	identify,	motion, 0.1,2		
sps,	1,	trans,	init			

Table 3-5: Observed Sequence of Signals

Monitored Item	Verification Type	Info derived from
State Info		
Model_state	info	model
stateFrom	vv	code
stateTo	cc	code
isCond	cc	code
isPseudoFrom	cc	code
isPseudoTo	cc	code
transType	cc	code
Distribution Info		
nodeId	vv	code
hostname	vv	code
hostId	vv	code
Message Info		
cmd	vv	code
sender	vv	code
InstS	vv	code
nodeIdS	vv	code
receiver	vv	code
instD	vv	code
nodeIdD	vv	code
channelSDL	info	code
channelLog	vv	code
channelPhys	vv	code
actLen	vv	code
dataMin	info	model
dataMean	info	model
dataMax	info	model, code
Timer Info		
timerOps	vv	code
timerMode	cc	code
timerSignal	cc	code
timerResource	cc	code
timerCyccles	cc	code
timeMin	info, cc	model, code
timeMean	info, cc	model, code
timeMax	info, cc	model, code
UDF call		
UDFcallee	vv	code
UDFlen	info	model
UDFname	cc	model
General Info		
processCurrent	vv	code
instanceCurrent	vv	code
format	info	code
now	vv	code
Model_line	info	code

Verification Type

- cc

  for cross-checking purposes only, but not for construction of the *Observed Model*.

- vv

  for verification of the code generator and construction of the *Observed Model*.

- info

  for additional information only

- model

  information is derived from model and inserted into source code for cross-checking

- code

  information is derived from executed code or the ISG run-time system.

Table 3-6: Recorded Properties

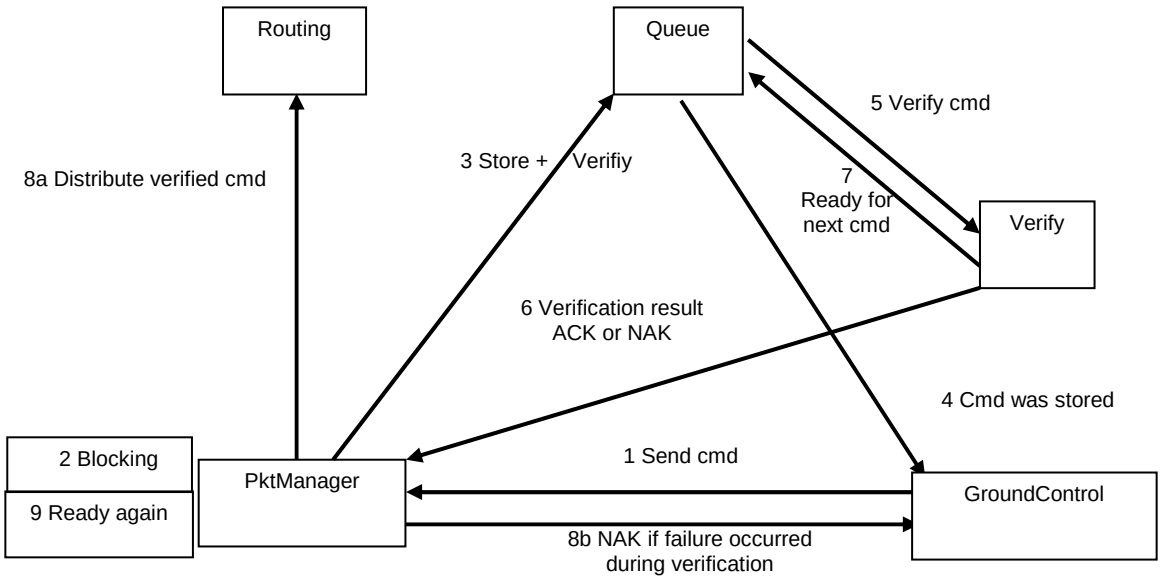

Fig. 3-1:The Application Mode

Timers may be created either with constant or varying periods randomly chosen from a given interval. Also, timeout conditions may be set when a reply is expected. Exception handling is required for every state, in order to manage messages not expected in the current state.

The communication scheme may be specialised for every instance of a process type, i.e. for sending of a message a list of receiving instances may be specified including specific terms like “return to sender instance”..

When the automatically generated code is executed a sequence of events is recorded like shown in Table 3-5. Its contents was simplified for better understanding. Such sequences are analysed and the retrieved information is converted back into code of the modelling language ISGL. Several different sequences observed for a process may contribute to obtain an equivalent to the original code. E.g. instances of a process may differ in communication rom each other, or the length of the data attached to a message may differ, so that several samples may be needed to conclude on branches in the model or a range. Similarly it is for timing figures, e.g. for the expiration deadline of a timeout.

4 FEEDBACK FROM PRACTICE

4.1 The Application

Fig. 3-1shows an application to which the back transformation was applied. This is the same application as discussed in a previous paper on model transformation [4].

The model represents a manager for telecommands which are received from ground and processed on-board through several stages (queuing, verification and routing) which all are controlled by a packet manager.

4.2 The Implementation

The back-transformation was implemented in several steps starting with evaluation of the sequence of actions executed during a state transition to construct the principal skeleton of the FSM of a process type.

Then dependencies on instances were identified and put into the observed model. In a following step the envelop of performance properties was evaluated by analysis of a number of recorded sequences.

This way multiple shapes of observed records are mapped back into a single piece of code with variants for instances or timing.

4.3 Getting Back the Full Model

The reconstruction of the model requires observation of properties related to each statement type in the code. Consequently, only such code of the observed model can be generated which corresponds to fully covered code in the reference model: full test coverage is required to get a 1:1 correspondence between both models.

If the observed model is not complete, then either the reference model includes dead code or the executed tests are not sufficient.

The approach only does verify such parts of the code generator, which are executed during testing or operation. From a pragmatic point of view this is what is needed because there is no need to verify the code generator for code not being executed . However, from a rigorous point of view it would be desirable to get all parts of the code into operation, so that the full model can be reconstructed.

As a consequence, successful verification of the code generator implies sufficient testing which should be based on auto-testing for reduction of test effort.

4.4 Ease of Change Management

The chosen approach for verification of the code generator does not constrain its maintenance. No expensive and delaying certification process is needed to get it into operation again after a change. After execution of the generated code the correctness of the maintained version can easily be proven. This gives high flexibility for immediate improvements and extensions of the code generator.

4.5 Independency of the Modelling Language

Though this verification approach is based on the modeling language ISGL and the related code generator, the back transformation is available to other modeling languages by model transformation [4], too.

If model transformation ti ISGL is supported for a modeling language, then a back transformation is possible,too, for every model defined in another modeling language.

5 RELEVANCE TO SDL AND MSC

This verification approach is similar to the one known in the world of SDL [5] and MSC [6], where MSCs (Message Sequence Charts) are created when the FSMs defined in SDL are executed. These MSCs are the "Observed Signals". Some SDL tools (e.g. ObjectGEODE [7] and SDT [8]) provide the capability to define a number of "reference sequences" of signals in MSC notation, which can be compared with the observed sequences. Then either a match or no match is found or a counter example.

From this point of view, the ISG process for verification of the code verifier is inverse to the SDL/MSC approach:

- SDL/MSC expresses the reference and observed specification in MSCs, i.e. sequences of signals, deriving the observed sequences from the executed FSMs
- ISG expresses the reference and observed specification in FSMs included in models, and transforms the observed sequence of signals (extended MSCs) – as derived from the executed FSMs – back into a model (FSM) representation.

In case of SDL/MSC the verification is performed on the level of sequences of signals, in case of ISG on the level of FSMs and models. Moreover, the a.m. tools did not derive the MSCs from the generated code, but from an independent, separate simulator of which the code is quite different from the later target code.

ISG also addresses the aspects of performance and distribution, which are not considered in case of SDL/MSC, because they are not supported in the SDL/MSC representation. Therefore CGverifier can derive a model (spawned by behaviour / FSM, performance and distribution) from the observed signals generated by the code intended to run on the target system.

The essential difference is that CGverifier does not make a comparison on the level of the signals, but on the level of the constructor of the signals. This is possible because states and the message contents are part of such signals, which may be considered as an extension of the MSC syntax. The benefit is that the number of comparisons shrinks drastically, because not all combinations of the reference sequences need to be compared with every observed sequence. Instead, the many sequences of signals have to be filtered and interpreted as action sequences of FSMs.

While in case of SDL/MSC the specification only consists of sequences of signals, not allowing to construct an "observed model" equivalent to the "reference model", the ISG specification extends this representation to a specification model, which is the cornerstone for the automated comparison on modelling level.

6 CONCLUSIONS

The implementation of CGverifier demonstrates that a back transformation from observed properties into the modelling domain and an automated comparison is feasible. This shall allow in future to apply the automated verification of the generated code w.r.t. the reference model in a more general manner.

The current exercise focused on behaviour, performance and distribution, the essential properties of a distributed real-time system. In a future step this approach should be extended to functional properties.

As ISG supports integration of code from different tools, the back transformation will presumably also be based on partitioned verification of the contributions from different tools.

An advantage of the described verification approach is that the generated code can automatically be verified at little expenses for every executable version derived from a model ensuring by real facts that the generated code corresponds to the reference model.

In order to get the full model back from generated code each branch must be executed. This requires full test coverage and even more than that when performance properties shall be retrieved. In consequence, automated testing is required, too, to keep the costs of such extensive testing low. The issue of back transformation supports the need for full testing of code and model, it is a pre-condition of success.

The activity described above was funded in part by the ASSERT project [2], an Integrated Project (IP CL004433) executed in the course of the 6th Frame Programme of the EU from 2004 to 2007. The ASSERT consortium consisted of about 30 European companies lead by ESA as prime.

7 REFERENCES

[1] Scade, Esterel Technologies, Toulouse, http://www.esterel-technologies.com

[2] ASSERT: Automated proof-based System and Software Engineering of Real-Time Systems, IP CL004433, http://www.assert-project.net

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

[4] R.Gerlich, D.Sigg, R.Gerlich: Model Transformation in Practice, Proceedings of DASIA'07 organised by Eurospace Paris, 29.05.-01.06.2007, Naples, Italy

[5] SDL, Specification and Description Language, Z.100, International Telecommunication Union, ITU, http://www.itu.int

[6] MSC, Message Sequence Charts, Z.120, International Telecommunication Union, ITU, http://www.itu.int

[7] ObjectGeode, SDL toolset, Verilog (now part of Telelogic)

[8] SDT, SDL toolset, Telelogic, http://www.telelogic.com