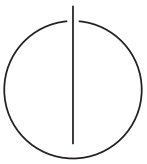
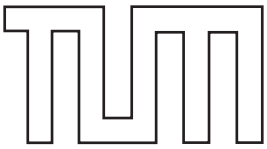


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Software & Systems Engineering
Prof. Dr. Dr. h.c. Manfred Broy



SPES 2020 Deliverable 1.3.B-5

Analysis Techniques: SWM Case Study and AutoFocus 3 Model Checking support

Author: Alarico Campetelli
Version: 1.0
Date: 4 January 2011
Status: Released

Technische Universität München - Fakultät für Informatik - Boltzmannstr. 3 - 85748 Garching

About the Document

A modelling theory should support appropriate analysis techniques in order to verify, test and validate the specified systems. In this document we analyse the properties to be verified, based on a case study in the energy domain. In this case study a prototype of a virtual power station is built using consistent model-based development. Furthermore, we present the current state of the implementation of the formal verification procedure in AutoFocus 3, founded on the work proposed in the SPES 1.3.A-2 deliverable.

The purpose of this document is to provide an introduction to the properties verifiable with the proposed analysis techniques, evaluate the actual implementation and propose the future work. The integration of such techniques into the modelling theory is one of the objective of work package ZP-AP 1.3.

Contents

1 Introduction

In the last years, the deployment of embedded software systems, which involve important human activities, it has raised interests about safety issues. A combination of fault prevention, fault tolerance, fault removal, and fault forecasting techniques are commonly used in order to achieve a high degree of reliability. We develop analysis techniques in SPES, founded on formal verification, this technique is exhaustive and usually improves the knowledge of system. However, it requires specific knowledges and is time-consuming, only as reliable as the formal models used. Formal verification tools are usually complex, nevertheless in the last years their use is increased, particularly in standard and formal development processes. We have presented in the SPES 1.3.A-1 deliverable [Cam09] a general introduction to analysis techniques, in industry and research.

In this document, we study the properties to be verified, by the techniques proposed in the SPES 1.3.A-2 deliverable [Cam10]. Indeed, in such deliverable [Cam10] we proposed for the tool AutoFocus 3 [SPHP02], based on Focus theory [Bro07], model checking solutions. We consider now the impact of such techniques in a real scenario and present the actual state of the implementation.

SWM Case Study In order to study the applicability of the model checking in the SPES framework, we have considered different possibilities to specify the properties to be verified. We report here a case study presented in the SPES energy domain, realised with the collaboration of Stadtwerke München (SWM). The goal of this case study is to construct a prototype of a virtual power station using model-based development. It will address both theoretical and practical limitations of the applicability of the tool in the development of a complex example, analysed in the energy domain. Another goal that is pursued with the prototype development, is to gain knowledge and experience on the development of virtual power plants.

The case study is interesting for the application of verification techniques. Our goal in this document is to study its requirements for defining the specifications to be verified by model checking.

AutoFocus 3 Model Checking We present the current state of the implementation of model checking techniques in AutoFocus 3. In fact we have added support for the SMV model checker, following our proposal in the SPES 1.3.A-2 deliverable. We describe information, details and some test results about this first support for SMV in AutoFocus 3. We summarise then the open questions and the future works.

Outline We resume our proposal for analysis techniques in SPES that we have presented in the SPES 1.3.A-2 Deliverable and the modalities in order to specify the properties for model checking, in Section 2. In Section 3 we describe the analysis of the case study and the resulting formal specifications. Then in Section 4 we report the actual implementation of the model checking support in AutoFocus 3 and finally we conclude in Section 5.

2 Analysis Techniques

In the Deliverables 1.3.A-2 we presented how we support analysis techniques in the SPES project. Based on this proposal we elaborate on the properties to be verified and on the model checking implementation details. In this document the modelling theory considered is FOCUS and AutoFOCUS 3 is used as support tool. AutoFOCUS is the modelling tool integrated in Eclipse, implemented for applying the FOCUS theory. It was arrived at the third release.

Our proposal is mainly based on formal verification, applied to AutoFOCUS 3 models and to *C*0 (a C language subset constructed for usage with the Isabelle/HOL [WPN08] verification environment as discussed in [Sch06]) generated from them. We opt for methods in the framework of model checking, but we also want to consider other formal verification techniques. Figure 1 represent our proposed methodologies.

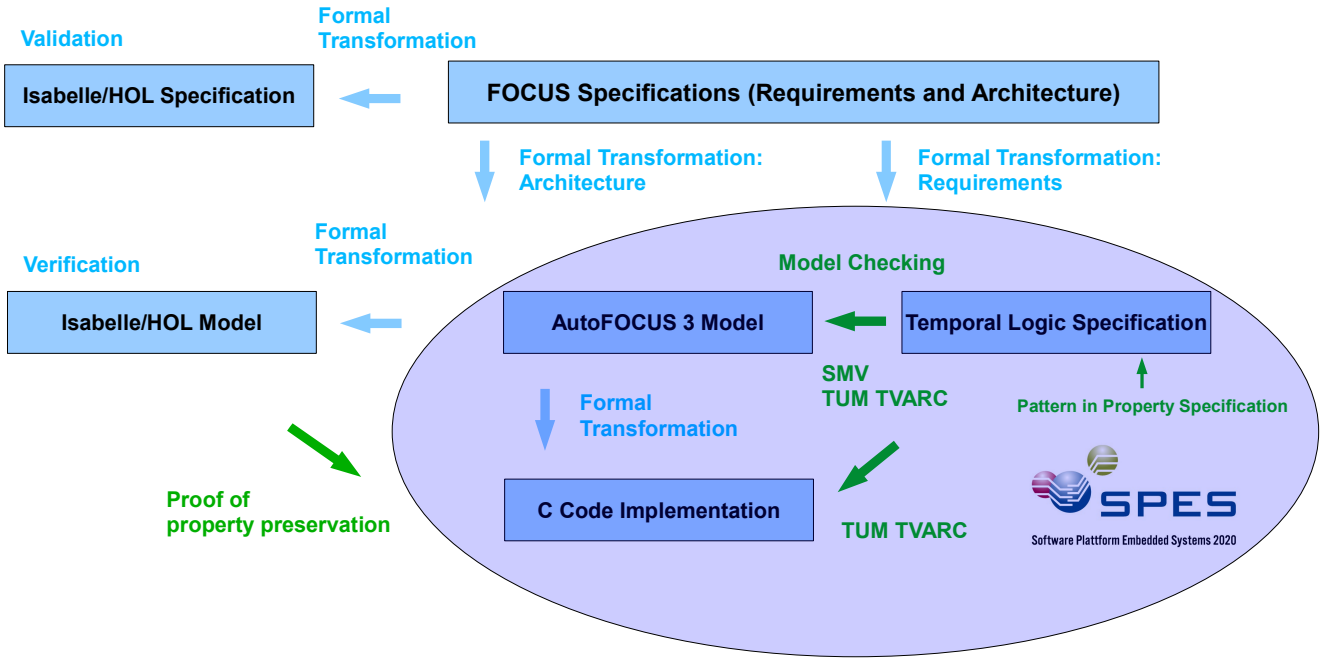


Figure 1: Overview of Analysis Techniques based on the Verisoft XT (http://www.verisoftxt.de) approach with the SPES proposal.

Our first proposal is based on model checking, but we also want to study other formal verification methods as for instance theorem proving [ASM+08], represented with the support to Isabelle/HOL in the Figure 1. In this document we analyse the properties to be evaluated through model checking, a case study for specifying such properties, verification test results and the actual model checking implementation in AutoFOCUS 3.

2.1 Model Checking

In short model checking is an automatic method which, given a model of the system to verify and a formal property, proves in a systematic way whether the model satisfies the property. One important characteristic is that the verification is exhaustive, i.e., all the system states

represented by the data values combinations are calculated, and in case the property does not hold a counterexample is usually provided.

Since all the possible input and states are considered in the verification, the model checking approach suffers from the "state space explosion" problem. However, model checking have made important progresses that have permitted to deal with very large state spaces; by using a symbolic representation of the state space, therefore without explicit enumeration of the states and by abstraction, practically removing information from the concrete system. Support tools have been successfully applied to very large state spaces in hardware and software verification.

The advantages of such techniques fit with the criticality of embedded systems development, which require an exhaustive and complete verification. We propose to use in SPES own model checker based on Three-valued Abstraction Refinement (TVAR) [BG99]. TVARC is based on a three-valued logic system with notions of abstraction and refinement. Besides our research solution for model checking, we proposed also the support for a well-established model checker: SMV (Symbolic Model Verifier) [McM92]. SMV is applied to the AutoFocus 3 models representation, while TVARC to the generated C0 code. The formal properties to be verified on such representations are specified in temporal logic [Pnu81], Computational Tree Logic (CTL) [CGP99] and Linear Time Logic (LTL) [Pnu81] for SMV and μ-calculus [Eme96] for TVARC, respectively.

2.2 Formal specifications for Model checking

We report here the different ways for specifying the properties to be verified by a model checker. We follow an order starting from the lower abstract level, if we consider temporal logic an assembler language in programming, meanwhile SALT [BLS06] and specification patterns [DAC99] as higher level language that are translated then to temporal logic.

Temporal logic Temporal logics, such as CTL and LTL, are specification formalisms where the propositions are qualified in terms of time. They are used to define with an exact semantics properties of a set of traces. Temporal logic specifications have been successfully verified by automatic verification techniques, such as model checking.

Structured Assertion Language for Temporal Logic (SALT) Temporal logic is considered to be a tool for skilled verification engineers and researchers, because of the difficulties in the formalism to write the properties. In order to create compact temporal specifications to be used in the formal verification, was defined the specification and assertion language SALT, that means Structured Assertion Language for Temporal Logic. SALT provides users with a specification language with a higher level of abstraction with respect to other formalisms as temporal logics. It is based on ideas of existing approaches, such as specification patterns but also provides nested scopes, exceptions, support for regular expressions and real-time. It has some constructions similar to a programming language and its specifications are more intuitive to utilise and understand than temporal logic. The actual implementation translates the SALT specification to LTL formulas. SALT supports the temporal operators of the temporal logic, but it provides sophisticated scoping rules, support for (limited) regular expressions, exceptions, iterators, counting quantifiers, and user-defined macros. We use SALT as a front end to the model checking tools proposed in SPES.

Patterns in Property Specifications In general formal specification and verification methods have not been widely adopted, partly due to the lack of definitive evidence in support of the cost-saving benefits of formal methods. There are also obstacles to adopting formal methods including lack of a good tool support, appropriate expertise, good training materials and process support for formal methods. Some of these barriers may be overcome by the availability of automatic tool for formal verification. Anyway, users of such tools, such as model checking, still must know the specification language of the tool, for specifying the properties. The acquisition of this level of knowledge may be an obstacle to the adoption of automated verification tools.

In [DAC99] the authors propose a pattern-based approach for the presentation, codification and reuse of property specifications for finite-state verification. We report the specification pattern definition: "A property specification pattern is a generalized description of a commonly occurring requirement on the permissible state/event sequences in a finite-state model of a system. A property specification pattern describes the essential structure of some aspect of a system's behaviour and provides expressions of this behaviour in a range of common formalisms." [DAC99].

Currently, in the formalisms supported there are amongst others LTL, CTL, μ-calculus, and other temporal logics. The patterns are based on a survey of available specifications, collecting over 500 examples of property specifications. Most of specifications are instances of patterns.

The pattern validity is specified by a scope, which denotes the extent of the program execution where the pattern must hold. There are five kinds of scopes: *global* for the whole program execution; *before* for the execution till a given state or event; *after* for the execution after a given state or event, *between* any part of the execution from one given state or event to another state or event; and *after-until* that is like between but the selected part of the execution continues even if the second state or event does not happen. In order to determine the scope the user has to specify a starting and an ending state or event for the pattern. Figure 2 describes for each scope the portions of an execution that are indicated by the scopes. The scopes should be understood as optional, in fact the specification will be true if the scope delimiters are not present in an execution.

The patterns are organised in an hierarchical way based on their semantics (Figure 3). For instance, some patterns necessitate that states or events have to occur or not, as the absence pattern, while other patterns require a specific order of states or event, as the response pattern. The organisation for the specification pattern distinguishes properties between properties that consider the occurrence or ordering of the states or events during program execution.

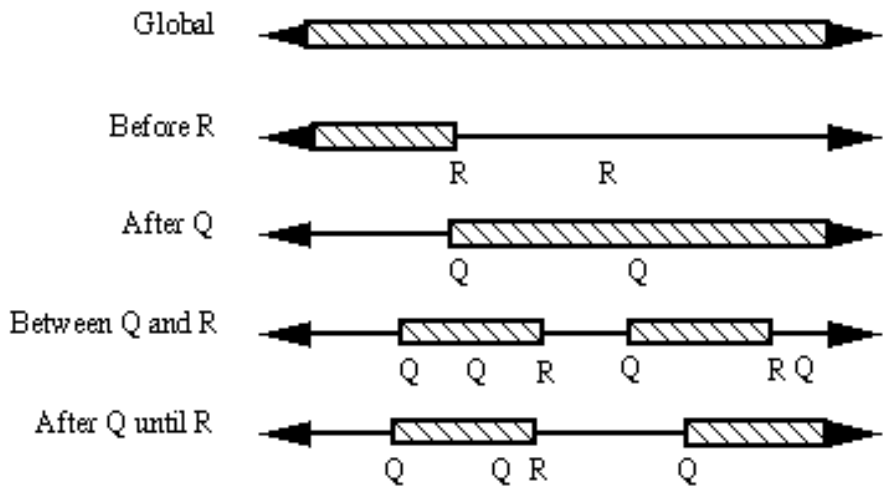


Figure 2: Pattern scopes. Image from the Specification Pattern Website (http://patterns.projects.cis.ksu.edu/)

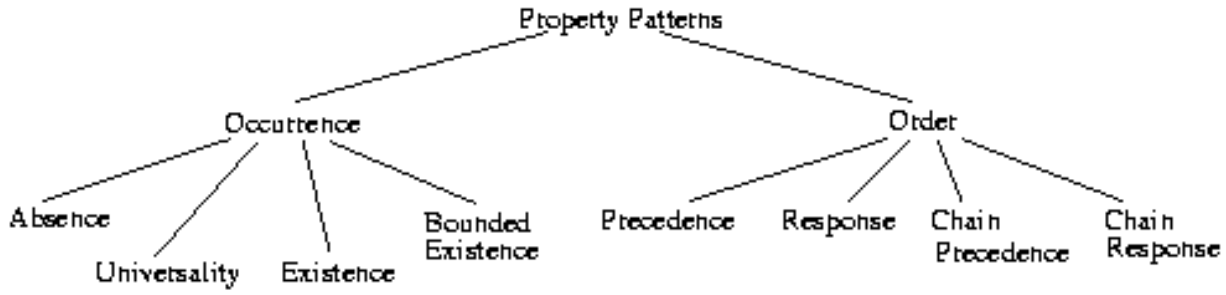


Figure 3: A Pattern hierarchy. Image from the Specification Pattern Website (http://patterns.projects.cis.ksu.edu/)

3 SWM Case Study

The goal of the SWM case study is to construct a prototype of a virtual power station using consistent model-based development. We provide an example of a properties specification, to be verified using model checking, based on this SPES case study. We have analysed the use case diagrams provided with the case study documentation, in order to derive the properties to be verified.

3.1 Use Case: Switch-on

The first use case is about the switch-on of the generating plant, used for generating the energy. We report here the use case:

- Actors: generating plant
- Abstract: The production plant is turned on and initialises its indicator
- Preconditions:
  1. A generating plant gets the signal to turn on (Q)
- Postconditions:
  1. The production system is switched on (P1)
  2. The display of the generating plant shows their current capacity, current active power generation(0), current required active power generation(0) and status (signed off) (P2)
- Normal run:
  1. The production plant starts up
  2. The production plant turns on its display
  3. The production plant arranges the following data:
     - their current capacity
     - their current active power generation (0 Watt)
     - their current required active power generation (0)
     - its current status (signed off)
- Alternative Run:
  1. see the normal run
  2. see the normal run
  3. If the production plant is a cogeneration power station (BHKW), it displays the following data:
     - their current capacity
     - the current heat consumed by the operator of the cogeneration power station (BHKW)
     - the current consumed active power by the operator of the cogeneration (CHP)

- their current active power generation
- their current available capacity for the virtual power plant (VKW) (0)
- their current required active power for the virtual power plant (VKW) (0)
- the current operator of the CHP by the network based active power
- its current status (signed off)

Properties This use case describes the expected behaviours and conditions when a generating plant is switched on. We consider the preconditions as exact conditions, which imply some system behaviours. The postconditions are such behaviours that must then become true. So an informal description for the property is:
Every time the generation plant is switched on the following actions must happen

- *The production system is switched on*
- *The production display is activated*
- *The production plan arranges the informations on the display*

This property can be modelled with a specification pattern. We show the hierarchical organisation for the pattern system in Figure 3. The major subdivision is between patterns in the framework of "occurrence" and "order". We can consider the "universality"pattern in the occurrence group, which describes a portion of a system's execution which contains only states that have a desired property. In this use case we need that the first statement is always satisfied. The universality pattern has the following form, with the general statement on P and five alternatives scopes:

Universality
P is true:
Globally
Before R
After Q
Between Q and R
After Q until R

The scopes are common to each specification patterns and are shown in Figure 2. We can model our property using the universality pattern with the "After Q" scope:

$$(P_1 \; and \; P_2) \; are \; true \; after \; Q$$

where the postconditions are *P1* and *P2* and the preconditions Q. It is possible to define this property with a specification pattern, then verify it through model checking without to learn or the knowledges of temporal logic notions. We have only to formalise the placeholders P and Q in order to define the property. The same property may be expressed with the assertion language SALT:

$$assert \; always \; (Q- > always(P_1 \; \& \; P_2))$$

In this case the property should be thought and constructed as a logical formula. It might be easy to construct the formula for developers, but in some domains it is not comfortable to

use logic operators. Finally we can express the property directly in a temporal logic, as LTL, accepted by SMV:

$$G\ (Q- > G((P_1\ \&\ P_2)))$$

In this case, it is even more difficult to write the property, one has to know which is the meaning of the temporal operators such as G (Globally).

3.2 Use Case: Adapt the operation mode of the plant

We consider now a second use case, where the operation mode of the plant is changed.

- Actors: production system, Virtual Power Plant
- Abstract: The virtual power plant adjusts the active power production of its affiliated production plants and shows the updated required active power production of the individual generating units.
- Preconditions:
  1. A generating plant has logged in, or (Q1)
  2. A generating plant has logged off, or (Q2)
  3. The capacity of a production system has been changed, or (Q3)
  4. The total power consumption has changed, or (Q4)
- Postconditions:
  1. The sum of the required active power of the individual constructions corresponds to the total consumption (P1)
  2. The updated required active power production of each generation plant will be displayed (P2)
- Normal run:
  1. The virtual power plant calculates the current active power to be generated.
  2. The virtual power plant calculates the most cost-effective operation mode of its plant. It takes into consideration, that certain production systems require a minimum active power production to be profitable, the different cost functions for the active power production in the different production systems and their current capacity.
  3. The virtual power plant sends to all generating units whose active power production must be adjusted, a new value indicating the active power to be generated by the production plant.
  4. The virtual power plant disposes to display the current required active power production of each generation unit.
  5. The production facilities adapt their active power production according to the signalled new required active power production from the virtual power plant.

We can use again the universality pattern *"P is true after Q"*, where the postconditions are P and the preconditions Q. We formalise the placeholder Q as the logical OR of the single preconditions, and for the P we model it with the logical AND between the two postconditions. For the second postcondition ($P2$) we use recursively the specification pattern "existence" with the "Globally" scope.

Existence
Q becomes true:
Globally
Before R
After S
Between S and R
After S until R

Therefore the pattern specification is:

$$(P_1 \; and \; P_2 \; existence) \; are \; true \; after \; ((Q_1 \mid Q_2 \mid Q_3 \mid Q_4))$$

We use SALT to express the same property:

$$assert \; always \; ((Q_1 \mid Q_2 \mid Q_3 \mid Q_4) -> always \; (P_1 \; \& \; eventually \; P_2))$$

Finally the correspondent LTL temporal logic formula generated is:

$$G \; ((Q_1 \mid Q_2 \mid Q_3 \mid Q_4) -> G(P_1 \; \& \; F(P_2)))$$

Also in this case one needs appropriate temporal logic knowledges for writing such formula without a specification pattern.

3.3 Specification pattern implementation

In the presented examples we use the specification patterns to define the variables placeholders. In AutoFocus 3 the user should know implementation details from the model, for specifying which constructions and values correspond to the precondition and postconditions.

Therefore in the implementation of the specification patterns in our modelling theory, we should also provide an easy way to find and select such information. It would be error-prone and not comfortable to write the placeholders direct by hand, and is even not possible if the user does not know the implementation and the translation usable by the SMV model checker. A reasonable solution is to provide, for instance, the list of all the variables, ports or other meaningful information of a component or process, with default values, operator suggestions and type information, integrated in the model checking interface.

We want to add a solution based on specification patterns in AutoFocus 3 for the model checking activity. It will be integrated with an user interface and therefore the properties are specified without specific temporal logic knowledges. This proposal might be used in an industrial context, where advanced knowledges about temporal logic and formal verification are not common.

4 Model checking in AutoFocus 3

We present the current state of the model checking integration in the tool AutoFocus 3. As depicted in Figure 1, our idea is to support at first model checking in the framework of the Focus theory and therefore in the AutoFocus 3 tool, and then consider other formal verification techniques. We propose the support for an existing model checker (SMV) and for a research solution (TVARC). They have different characteristics and are applied to different abstraction levels of the modelling theory.

4.1 SMV and TVARC

The verification with SMV is applied directly to the AutoFocus 3 models. The initial implementation added such support for the AutoFocus 3 components and processes. The users can specify a property in an interface window and then execute the verification. When it is executed the verification, the system then translates the AutoFocus 3 models in an SMV instance and apply the model checker for proving its validity. Additionally, we want to apply the TVARC model checker to the generated C0 from the AutoFocus 3 models. Figure 4 illustrates these concepts.

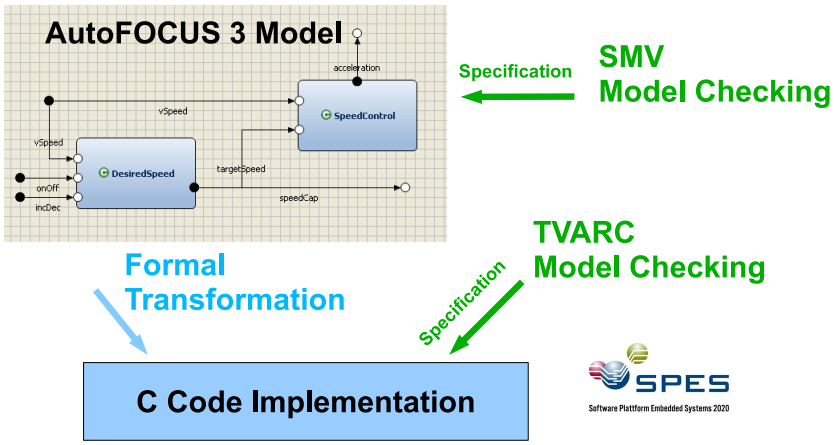


Figure 4: Initial model checking implementation

4.2 SMV integration

We have realised an initial integration for the SMV model checker support for AutoFocus 3. We hierarchically translate the AutoFocus 3 components and processes to an SMV instance. It is provided a configuration page where the user can define the specification to be verified in LTL, CTL or SALT. In this page the user can also specify the value interval for the integer variables, in which SMV will perform the verification and other configuration parameters. It is important for users to handle the integer interval, so that they can select the right range

of values for the verification and also avoiding the state space explosion problem with large intervals.

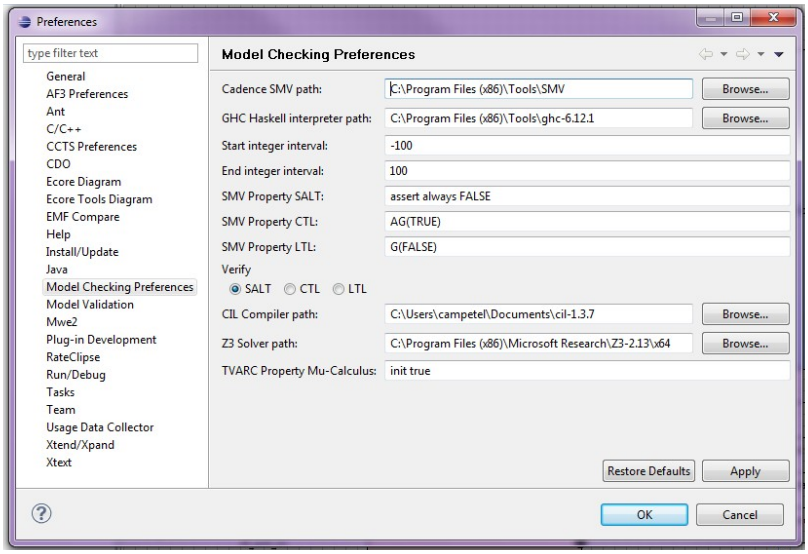


Figure 5: Preference page for model checking in AutoFocus 3

In Figure 5 there is the configuration interface for the model checkers in AutoFocus 3. Future improvements might be to render the interface more comfortable, especially for editing the logic formulas. It would be convenient to have for instance the list of the variables or other details in the same interface or also the implementation of specification pattern interface, as we argued in Section 3.

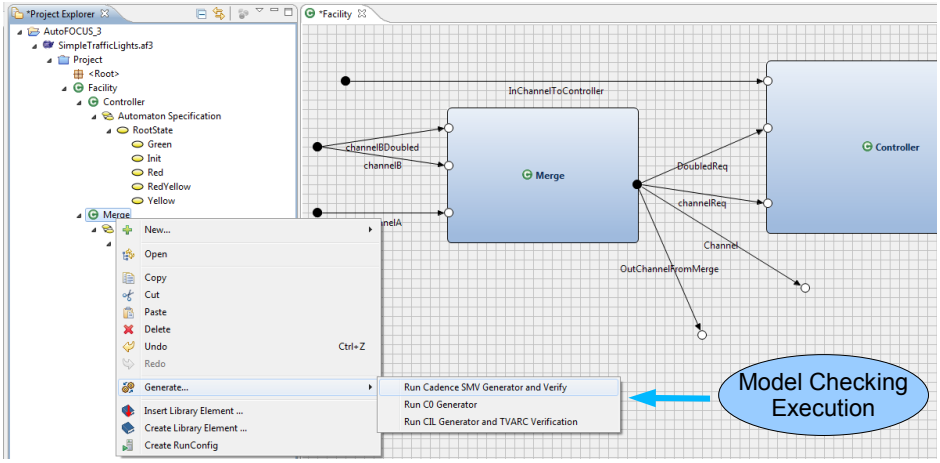


Figure 6: Interface for executing Model checking in an AutoFocus 3 model

In Figures 6 and 7 there are the screenshots of the application of SMV model checking in the tool AutoFocus 3, where the relative options are correctly set. In Figure 6 there is an example of the selection of a component (or process) in which we want to apply the verification, provided by the relative model checking option. The component will be then automatically translated with all its subcomponents, in a SMV instance where each state machine is represented by

an SMV module and hierarchically a module contains the calls to the modules of the relative subcomponent. Then the verification is executed and the result will be presented to the user as shown in Figure 7.

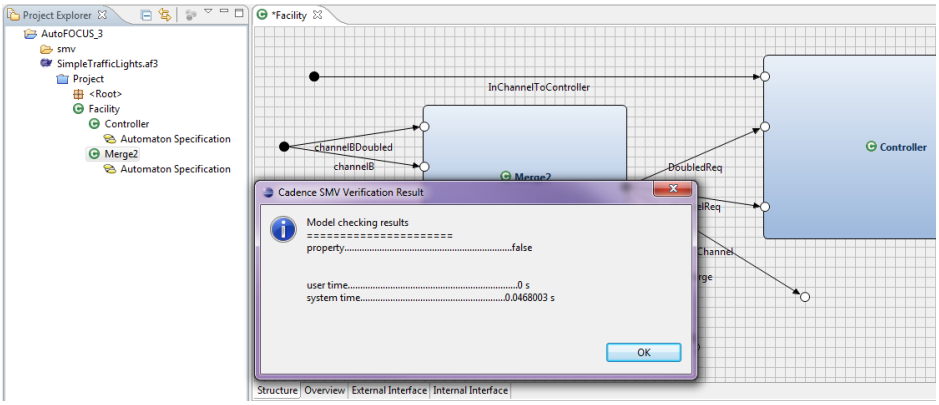


Figure 7: Model checking result in AutoFocus 3

4.3 SMV verification limits

In the construction of the SMV model from the AutoFocus 3 model, we have restrictions for recursive constructs. In fact, in AutoFocus 3 there are recursive constructors, functions and data types, which our translated SMV model does not support. We can't verify SMV instances with recursive construct defined, in this case SMV won't return a correct result. This limitation is due to the finite data system of SMV, which has problems to handle infinite structures. A solution is to apply abstraction techniques for such constructs, anyway in a first implementation we return a warning message to the user, when he tries to convert an AutoFocus 3 model with such characteristics.

Another limitation is that the integer interval must be specified from the user. We noticed that also for relative small projects, with big integer intervals the verification process becomes too complex and so it might require too much time and memory. A solution is to implement integer abstraction techniques for SMV, but there is no easy and elegant solution to implement it.

Anyway, with the support for TVARC it will be possible to test the C0 generated code, with a bigger integer interval with respect to SMV. We can in fact overcome the integer interval limit, with TVARC, which abstracts during the verification the integer values in a more effective way.

4.4 TVARC integration

We apply the TVARC model checker to the generated C0 code as depicted in Figure 4. We propose to use this model checker, which is based on Three-valued Abstraction Refinement (TVAR) because it abstracts integer values during the verification procedure in an efficient way. With the support for TVARC, which is currently integrated into AutoFocus 3, it will be possible to test the generated code, against μ-calculus properties (Figure 5), without manually limiting integer variables. Therefore, we can double-check the specified properties and reach an even greater level of confidence.

4.5 Model Checking Case Study

We have tested the SMV implementation in AutoFocus 3, with an industrial case study modelled for an automotive project developed by Technische Universität München and a company that work in the automotive sector. This project advances the methodology for the model-based development of automotive systems and presents an application of the advanced methodology to a case study.

We have verified an AutoFocus 3 model, which represents an automotive hardware, more precisely some properties that were specified as requirements in the project documentation. In the verification of the model a data type was to complex to be verified, so we reduced the number of variables of such type so that the model checker was able to verify it.

4.6 Future Work

We are testing our SMV implementation with different AutoFocus 3 model instances, in order to consider the performance in terms of memory and time. In some industrial examples we have successfully tested some property in validation and falsification. It will be interesting to compare the SMV results with TVARC. Our feeling is that TVARC might work better in terms of memory occupation because it uses a better data abstraction. On the other hand it might be slower also for the complexity of the abstraction used. Some preliminary tests seem to confirm our impressions.

Case Study and Integration The next step is to find a case study for confronting both model checkers in AutoFocus 3. It will be also necessary to improve the interface for the specification of the properties and also the global integration of their support. A further implementation is also to apply TVARC directly to the AutoFocus 3 models, without the need to use the generated C0 code.

Specification Patterns We observed in Section 3 that the properties specified direct in temporal logic or by SALT require knowledges logic that may be not common between the AutoFocus 3 users. We propose to implement specification patterns in order to to easily specify the properties to be verified. We want to apply an implementation of specification patterns, based on the thesis work "Investigating the transition from informal, functional requirements to SALT formulae". This work studies the possibilities for defining informal specifications, which are then translated in formal ones. It is provided a graphical interface, for defining the patterns. Then the properties are translated in a formal specification for the model checker. Our idea is to provide this specification patterns interface in AutoFocus 3.

Counterexample handling A model checker checks if the system satisfies the property and gives a "yes" or "no" answer. If the answer is no, i.e., the system does not satisfy the property, model checkers usually output a counterexample, i.e., a run of the system which violates the property. The counterexample can be analysed to discover bugs in the system design. Besides such integration, a future improvement would be the implementation of counterexample analysis. The information provided by the counterexample might be related to the model and presented in a graphical way in AutoFocus 3, in order to be useful for the debug phase. We are interested to use the MSC editor integrated in AutoFocus 3, for tracing the information provided from the counterexample and also execute a simulation of them.

Theorem Proving We have some limitations in the verification with SMV, as the infinite data constructions and the integer interval. We might overcome this limitation with TVARC, because it has a better integer abstraction. Nevertheless we consider also the possibilities for other formal verification methods, as theorem proving.

Theorem proving utilises formulas in some mathematical logic or theory, to express the system and its properties [CW96]. A formal system based on this logic, defines a set of axioms and a set of inference rules. In practice, theorem proving finds from the axioms of the system a proof of a property. The proof is composed of steps which invoke the axioms and rules, and possibly derived definitions and intermediate lemmas.

In deliverable 1.3.A-2 [Cam10] we described the existing theorem proving approaches for FOCUS and AutoFOCUS. Considering the model checking limitations in some verification problems, we are interested to study the work already done in the support for theorem proving in AutoFOCUS 3, and so evaluate if it might be used in the SPES framework.

5 Conclusion

In the first deliverable 1.3.A-1 we have reported the major analysis techniques used in research and industry, meanwhile in the second deliverable 1.3.A-2 we presented a first formal verification proposal for SPES, that is model checking. We have now mainly focused in this document on the application of model checking techniques within the modelling tool AutoFocus 3.

We explained the most used techniques for modelling the specifications to be verified, in model checking, also with a case study. This case study is in the energy domain and is realised in collaboration with SWM. In order to derive the specifications we analysed use case diagrams, and from the results we evaluated the different specifications techniques, namely temporal logic, the assertion language SALT, and the specification patterns. We conclude that the most high level method, that is specification patterns, is more comfortable and understandable for the users. Anyway the possibility to define the properties also with the other formalisms, should be guaranteed for all the users.

We presented also the first support for model checking in AutoFocus 3, alongside with its verification limits. We have tested the model checker with an industrial case study and we were able to verify some project requirements. Based on such studies we propose some future work for the analysis techniques in SPES.

References

[ASM+08] Rev R. Acad, Cien Serie, A. Mat, Matt Kaufmann, and J Strother Moore. Ciencias de la Computación / Computational Sciences Some Key Research Problems in Automated Theorem Proving for Hardware and Software Verification. 2008.

[BG99] Glenn Bruns and Patrice Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In *11th International Conference on Computer Aided Verification (CAV'99: Proceedings)*, pages 274–287, London, UK, 1999. Springer.

[BLS06] Andreas Bauer, Martin Leucker, and Jonathan Streit. Salt - structured assertion language for temporal logic. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775. Springer, 2006.

[Bro07] Manfred Broy. Two Sides of Structuring Multi-Functional Software Systems: Function Hierarchy and Component Architecture. In *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA'07, Proceedings)*, pages 3–12. IEEE Computer Society, 2007.

[Cam09] Alarico Campetelli. SPES 2020 Deliverable D.1.3.A-1: Analysis Techniques: State of the Art in Industry and Research. Technical report, 2009.

[Cam10] Alarico Campetelli. SPES 2020: Analysis Techniques suitable for SPES project. Technical report, Technische Universität München, 2010.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 1999.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996. Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[Eme96] E. Allen Emerson. Model checking and the mu-calculus. In Neil Immerman and Phokion G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. American Mathematical Society, 1996.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[Pnu81] Amir Pnueli. A temporal logic of concurrent programs. In *Theoretical Computer Science 13*, pages 45–60, 1981.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technical University of Munich, April 2006.

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-Based Development of Embedded Systems. In Jean-Michel Bruel and Zohra Bellahsene, editors, *Advances in Object-Oriented Information Systems (OOIS 2002 Workshops, Proceedings)*, volume 2426 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2002.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *21st International Conference on Theorem Proving in Higher-Order Logics (TPHOLs 2008, Proceedings)*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer, 2008.