TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

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

SPES 2020 Deliverable 1.3.A-2

Analysis Techniques suitable for SPES project

Software Plattform Embedded Systems 2020

Author:	Alarico Campetelli
Version:	1.0
Date:	January 11, 2010
Status:	Released

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

About the Document

In SPES project we have introduced the Focus modelling approach for the development of reactive systems. The purpose of this document is to provide the actual research status of analysis techniques that can be integrated and used in Focus. We present methods for verifying and validating model definitions and requirements by formal verification.

We list formal techniques based on model checking, simulation, and the theorem proving, which have been studied in the Verisoft XT project[1]. In the framework of SPES project we propose mainly model checking techniques, considering the research work already done. The model checker proposed is our own solution based on the notion of abstraction and refinement of the model to verify. The integration of analysis techniques into the modelling theory is one of the objectives of work package ZP-AP 1.3.

[1] http://www.verisoftxt.de

Contents

1 Introduction

In the last years, the deployment of embedded software systems has raised many 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. Anyway there are no standard methods for applying analysis techniques of such systems. In fact, every industry, considering different application fields, tends to use their own particular development methodologies and so techniques for analysing and enhancing reliability.

We have reported in the SPES Deliverable 1.3.A-1 [Cam10], major analysis techniques and their state in research and industry. The techniques that we have analysed are variegated, and with different characteristics and purposes. Therefore the choice of the solution is made offsetting these characteristics and the needs of the project. We want to evaluate now which methods might be used in a modelling theory for SPES project. In our opinion the use of a formal verification [CW96] techniques can improve the overall reliability of the systems. With such techniques the whole development process can be made in an efficient way, using less time and so with economic saves. Formal verification performs the verification in an exhaustive way, provides a formal proof of the system's correctness and improves the knowledge of system. However, it is difficult and time-consuming, as reliable as the formal models used and it's difficult to prove the proof correctness.

In order to analyse the behaviour and the functionality of a system by formal methods, also the specification should have a rigorous definition, so it is not enough to use only formal verification. Actually in the industry formal methods are used seldom, despite the increasing of the use in international guidelines and standard for the development of safety-critical system. For the industries to implement formal methods in the whole life cycle development actually may need to much resource investments. So industries prefer to use formal verification within the existing development process. In the SPES project we want to provide a complete formal modelling theory, including formal verification techniques.

Overview Analysis Techniques in Focus In [HHR09] is introduced the Focus [Bro07] approach for the development of reactive systems. One of the goals of SPES project is to provide a complete formal modelling theory and therefore we aim to integrate formal verification. The aim of the current paper is to provide an overview over analysis techniques suitable for Focus specifications. We present the methodologies considering the results reached within Verisoft XT project[2]. In the framework of Verisoft XT are described verification techniques for embedded systems, with technical issues as well as case studies on embedded control systems. They have considered the verification of hardware, operating systems, and application software. The modelling theory used is Focus meanwhile AutoFocus 3 [SPHP02] is used as support tool. The verification is made from an informal specification through multiple transformation steps to obtain a verified formal specification. This research work is focused on appliance of simulation, model checking and theorem proving methods [CW96].

In this work we define the ideas for implementing analysis techniques in Focus, in two different levels. First, we propose to validate and verify the formal models translating them

[2]http://www.verisoftxt.de

for Isabelle/HOL theorem prover [WPN08]; Second, verification of the models and its corresponding C implementation code with a model checker. The specification to be verified is obtained from formal and informal requirements. With regard to the model checker we propose a well-know model checker and our own solution, based on actual research work.

Model Checking In few words model checking is an automated technique which, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) the model. One important characteristic is that the verification is exhaustive, i.e., all possible input combinations and states are taken into account, and a counterexample is usually generated in case a certain property does not hold. Model Checking is fully automatic, if we compare with other verification techniques, as for instance theorem proving. Verification is usually carried out by using algorithms to demonstrate the satisfiability of properties formalized as logical formulas over the model of the system.

Model checking approach suffers from the "state space explosion" problem, since (are considered) all the possible inputs and states during the verification. Recent technological advances, however, have managed to deal with very large state spaces by using a symbolic representation of the state space, therefore there is no explicit enumeration of the states; and by abstraction techniques, practically removing information from the concrete system. Such tools have been successfully applied to very large state spaces in hardware verification. We have considered model checking techniques for embedded systems also because such systems use fault tolerance, i.e., the systems guarantee safety properties, through redundancy. Model checking techniques can exploit such redundancy in our verification algorithms.

Therefore its characteristics, such as the use of redundancy, might help to reduce the state space explosion problem applying abstraction techniques.

An important technique to reduce the state explosion problem is in fact abstraction, where a smaller abstract model is verified that keeps an approximation of the behaviours of the original system. The major problem in applying this technique is in constructing such an abstract model. A successful technique is counterexample guided abstraction-refinement (CEGAR) [CGJ$^+$00] that addresses this problem by constructing abstractions automatically by starting with a coarse abstraction of the system, and progressively refining it, based on invalid counterexamples seen in prior model checking runs, until either an abstraction proves the correctness of the system or a valid counterexample is generated.

Our Model Checker In the same framework of abstraction and refinement for model checking we propose for AutoFocus specifications our own technique, which is based on three-valued logic [CGLT09]. Our Three-valued Abstraction Refinement (TVAR) approach extends the existing work on the three-valued model checking [BG99].

Outline We present the analysis techniques applied to Focus specifications and its layers in Section 2. In Section 3 we describe model checking and in Section 4 our model checker solution is presented. Future work are discussed in Section 5 and finally we conclude in Section 6.

2 Overview Analysis Techniques in Focus

We know that many embedded systems are used in safety-critical environments and so it is important to apply exhaustive verification and validation. Our proposal is based on formal verification techniques. A full systematic testing of such applications is often not possible because it is too time consuming or too expensive. However, errors or faulty states in these systems may lead to money or even lives losses. A replacement of the software while in operation is often too difficult or not possible for the nature of such systems and anyway it is not simple as in common desktop software. So the interest from the industries for formal verification is grown and several research projects have produced tools, but they are still not widely used and are not a standard in the development of embedded systems, also due to the needs of specific skills and knowledges in order to apply formal verification. Another factor that has limited the diffusion of such technique is also the difficulty to provide a standard tool considering that most of available tools use a proprietary input model. There are for instance few model checking tools that provide interfaces to standard development tools, as Statemate, MATLAB/Simulink etc or that are integrated into such tools for instance the SCADE Suite.

Our idea is to provide a complete modelling theory with verification techniques integrated. We have introduced for SPES the Focus modelling theory, now we describe the different layers that compose a model specification, and which analysis techniques to apply on each of them. The layers analysed are: Focus, AutoFocus and C code. Figure 1 contains a draft of the methodologies described in the next sections.

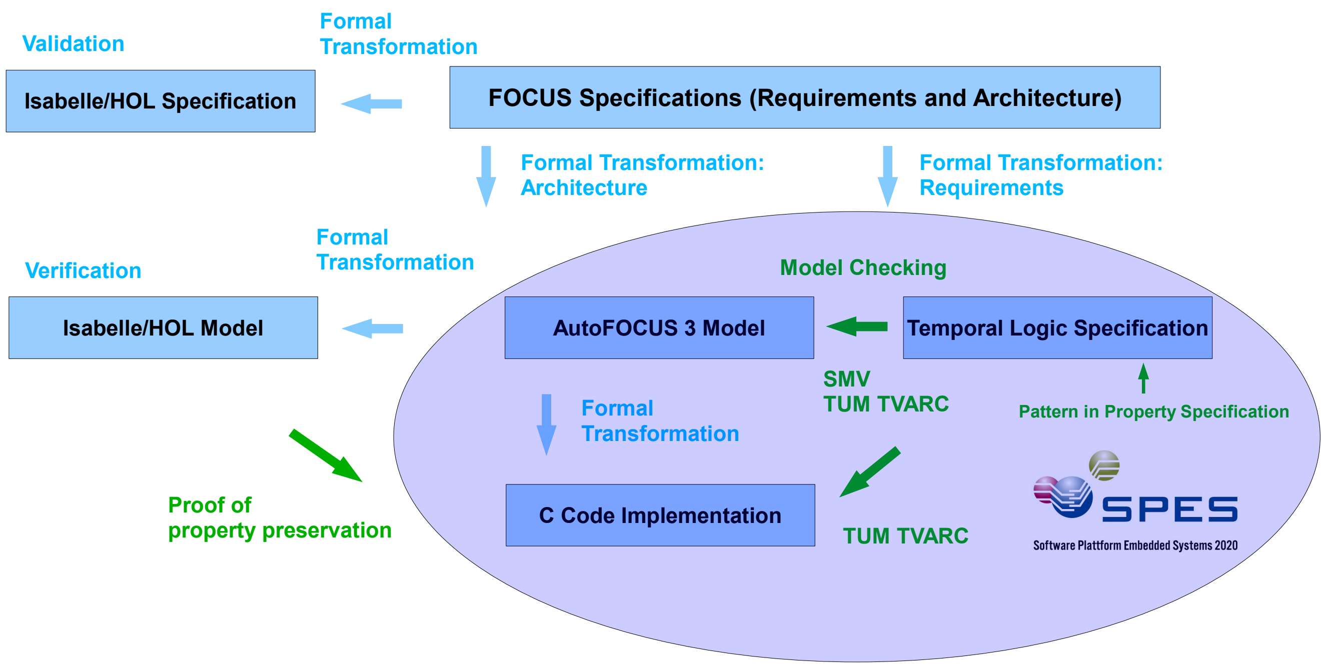
Validation
Formal Transformation
Isabelle/HOL Specification
FOCUS Specifications (Requirements and Architecture)
Formal Transformation: Architecture
Formal Transformation: Requirements
Verification
Formal Transformation
Model Checking
Isabelle/HOL Model
AutoFOCUS 3 Model
Temporal Logic Specification
SMV
TUM TVARC
Pattern in Property Specification
Formal Transformation
Proof of property preservation
C Code Implementation
TUM TVARC
SPES
Software Plattform Embedded Systems 2020

Figure 1: Overview of Analysis Techniques based on the Verisoft XT approach with the SPES proposal.

2.1 Focus layer

Focus is a framework for formal specifications and development of distributed interactive systems. This framework has an integrated notion of time and modelling techniques for unbounded networks, provides a number of specification techniques for distributed systems and concepts of refinement. Moreover, Focus specifications are much more readable and manageable, in fact the advantage of graphical notation is extremely important when we are dealing with systems of industrial size. Focus supports a variety of specification styles, which describe system components by logical formulas or by diagrams and tables representing logical formulas.

Theorem Proving In general we represent in Focus two kinds of specifications: a specification of system requirements and its architecture specification (corresponding to the black and the glass box view on the system, respectively). In Verisoft XT approach these representations prepare the ground to verify the system architecture specifications against the system requirements by translating both to the theorem prover Isabelle/HOL via the framework "Focus on Isabelle" [Spi07], and the refinement relation between them can be validated. The case when one needs to prove a single property of a system specification can also be seen as a refinement relation: this property can be defined as a Focus specification itself and then one just needs to show that the system specification is its refinement. Using this approach one can perform automatic correctness proofs of syntactic interfaces for specified system components. Verifying Focus specifications using Isabelle/HOL means proving that the semantics of the system architecture specification (white-box behaviour) implies the semantics of the system requirements specification (black-box behaviour).

2.2 AutoFocus 3 layer

We have described theorem proving for verifying and validating the Focus layer, but in some cases inconsistencies can still remain in the specification, model or code even after verifying certain properties. Thus, not only theorem proving techniques, but also testing, model checking, and simulation must belong to the development process. In order to apply such techniques in Verisoft XT approach the Focus architecture specification is translated to a representation in the related CASE tool AutoFocus 3[3]. AutoFocus 3 is a scientific research prototype tool, implementing a modelling language based on a graphical notation and a restricted version of the formal Focus semantics. This tool provides simulation and model-checking facilities.

Model checking In AutoFocus 3 functional properties can be specified using temporal logic notations, especially LTL (Linear Temporal Logic). Temporal properties can be checked using model checking tools, e.g., SMV (Symbolic Model Verifier) [McM92]. That way, system properties expressible in LTL can be verified by exporting AutoFocus 3 models to SMV and model checking the corresponding temporal formulas, as has been performed, for instance, for selected safety-critical properties in [FFH+09].

[3]http://af3.in.tum.de/

Besides the functional properties we also want to verify the Focus requirements specification. We can translate it or another kind of specification to temporal logic and again using the AutoFocus 3 model exported for SMV, we can check these temporal formulas. The transformations from Focus to temporal logic and to the AutoFocus 3 representation are demonstrated in a formal and schematic way in Verisoft XT approach. In this work we present our own TVAR model checking solution (described in Section 4). We propose as future work (Section 5) to use TVAR for verifying the AutoFocus 3 model exported in a representation for TVAR and the properties expressed by temporal logic, as already made for SMV.

Theorem Proving Within Verisoft XT project a code generator has been developed for creating Isabelle/HOL representations to prove the AutoFocus 3 models, as described in [Tra09]. We can formulate properties using more powerful notations expressible in HOL. Therefore a property is formulated as an Isabelle/HOL theorem and proved for the Isabelle/HOL representation of the system.

Simulation We can validate the model using the AutoFocus 3 simulator to get a first impression of the system under development and possibly find implementation errors that we introduced during the manual transformation of the Focus specification into a AutoFocus 3 model.

2.3 C code layer

As illustrated in [Höl09] the final product of the development process is C code generated from the AutoFocus 3 models. The AutoFocus 3 model is transformed to a corresponding C code by a code generator. More precisely a generator for C0 code, a C language subset constructed for usage with the Isabelle/HOL verification environment as discussed in [Sch06]. It is shown in [Höl09] that this transformation step preserves the properties of the model. C0 differs from C by restricting the language. Well-known, hazardous features, like pointer arithmetic, are forbidden in C0, while other restrictions, like the non-nested use of function calls, ease the reasoning and verification with Isabelle/HOL. The C0 code generated from AutoFocus 3 models does not need language features still available in C0 like dynamic memory allocation, pointers or arrays.

Model Checking We want to verify the C0 code representation with model checking techniques for a further verification of the implemented model. This approach is already applied in other model checkers based on C source code with applicability to embedded software. We propose our own TVAR model checker (Section 4) to make this verification step. We are working to prove the support to C0 programs, which will be completed in future work (Section 5).

Theorem Proving The C0 code generation is performed by the AutoFocus 3 tool, more precisely by a Java program that has not been formally verified. Therefore we cannot formally guarantee the correctness of the C0 code, even after the correctness has been shown for the AutoFocus 3 model and/or its Isabelle/HOL representation. In order to obtain this formal guarantee, the behavioural equivalence of the Isabelle/HOL representation of the AutoFocus 3 model and generated C0 code can be proved in Isabelle/HOL. The construction of a verification environment for C0 code in Isabelle/HOL was presented in [Sch06]. Such a verification environment can also be used for directly proving system properties for the generated C0 code.

2.4 SPES Domain

In the framework of the SPES project we consider systems which are implemented from specifications to AutoFocus 3 models. Our proposal is therefore based on verification techniques in the layers AutoFocus 3 and C0 code as shown in Figure 1. We aim to implement model checking and simulation methods.

3 Model Checking

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. Roughly speaking, the check is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite. The technical challenge in model checking is in devising algorithms and data structures that allow handling large search spaces. Model checkers, i.e., tools which perform model checking take two inputs: a finite state model of the system and a property specified formally. A model checker checks whether 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 also 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. Figure 2 describes the process of model checking. Model checking has been used primarily in hardware and protocol verification [CK96], the current trend is to apply this technique to analyse specifications of software systems.

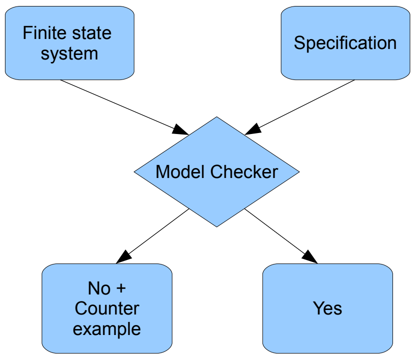

Figure 2: Overview of Model Checking

The specification to verify is usually expressed by temporal logic. Temporal logic is a class of modal logic, which extends propositional logic to incorporate time operators, in the sense that formulas can evaluate to different truth values over time. There are different types of temporal logic notations that correspond to different views of time (branching vs. linear, discrete vs. continuous, past vs. future, real time). Examples of temporal logic formal languages are μ-calculus, Linear Temporal Logic (LTL), Computational Tree Logic (CTL) and Timed CTL. Typically, the system is abstractly represented by a state machine, where the nodes represent the system's states and the arcs represent possible transitions between the states. Because state machine alone might be too weak to provide a complete and interesting description, the states and the transitions are annotated with more specific information. A common approach is to use Kripke structures [CGP99].

3.1 Kripke Structure

The Kripke structures are widely used in model checking, the semantics of temporal theories are traditionally defined in terms of Kripke structures. A Kripke structure is basically a graph having the reachable states of the system as nodes and state transitions of the system as edges. A labeling function maps each node to a set of properties that hold in the corresponding state. Kripke structures can be seen as describing the behaviour of the modeled system in a modelling language independent manner. Therefore, temporal logics are really modelling formalism independent; the definition of a state's properties is the only thing that could need to be adjusted for any formalism.

3.2 Properties verified

Temporal logic is a powerful tool to express several properties about systems. Examples of properties that can be expressed and so verified by model checkers are:

- Reachability: some particular situation can be reached
- Safety: something wrong never occurs
- Liveness: something desirable will ultimately occur
- Fairness: something shall (or shall not) occur infinitely often
- Deadlock-freeness: the system can always evolve to a successor state

While a violation of a safety property can be detected by a finite sequence of execution steps in the system, a violation of a liveness property may be detected only by an infinite execution of the system, practically finding a loop where the property is never reached. Consequently, liveness properties are harder to be verified by model checkers. The temporal logics listed have different ways to express these properties, and some property cannot be expressed in some logics.

3.3 State space explosion

The size of the systems to verify grows exponentially with the number of parallel system components, which limits the applicability of model checking techniques. In the last years one of the main goals of the research has been to extend the size of systems that can be effectively model checked. This problem is the major obstacle in applying model checking to industry. We list two approaches used to limit such problem.

3.3.1 Symbolic Model Checking

Symbolic Model Checking [McM92] was introduced around 1987 and is considered as an important breakthrough in model checking. Using this technique the representation of the system to verify are reduced, in order to handle the state space explosion problem. This technique utilizes a symbolic representation for sets of states and state transitions, which are represented by a Boolean encoding. The size of this encoding is greatly reduced by representing such structures by Binary Decision Diagrams (BDDs) [Büc62], which are traditionally used to represent Boolean functions. The verification for symbolic model checking is made using fixed point operators, which operate on the BDDs. The introduction of symbolic model checking was very successful in the last decade in the research community as well as in industry application especially in the verification of hardware designs.

3.3.2 Abstraction

Another improvement with respect to classical model checking algorithms is when only a restricted part of the whole state space is explored. An important approach consists of reducing the size of the model to be verified by abstraction, therefore replacing the concrete model of the system with a smaller abstract model. Abstract interpretation [CC77] is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems. Traditionally, abstraction techniques are designed to be conservative for property that hold. Thus, if a property is verified in the abstract system, it also holds for the concrete system. If, however, a formula does not hold for the abstract system, this may be because too much information has been hidden; say the abstraction is too coarse. In this case the abstraction needs to be refined, adding information which avoids having again the same spurious counterexample, i.e. a counterexample for the abstraction but that is not a valid counterexample in the concrete system.

3.3.3 Counterexample guided abstraction refinement

One of the successful abstraction technique is Counterexample Guided Abstraction-Refinement (CEGAR) [CGJ$^+$00]. The paradigm of iterative abstraction and refinement has gained momentum in recent years as a particularly effective approach for the scalable verification of complex hardware and software systems. CEGAR is an efficient automatic refinement technique which uses information obtained from erroneous counterexamples. The initial abstraction and the refinement steps are efficient and entirely automatic. The refinement procedure is guaranteed to eliminate spurious counterexamples while keeping the state space of the abstract model small. The advantages of this methodology have been demonstrated by experimental results and the interest received in research and application.

The CEGAR loop This technique constructs an abstract model of the system, and then a traditional model checker is used to determine whether properties hold in the abstraction. If the answer is yes, then the concrete model also satisfies the property. If the answer is no, then the model checker generates a counterexample. Since the abstract model has more behaviours than the concrete one, the abstract counterexample might not be valid in the concrete system, viz a spurious counterexample.

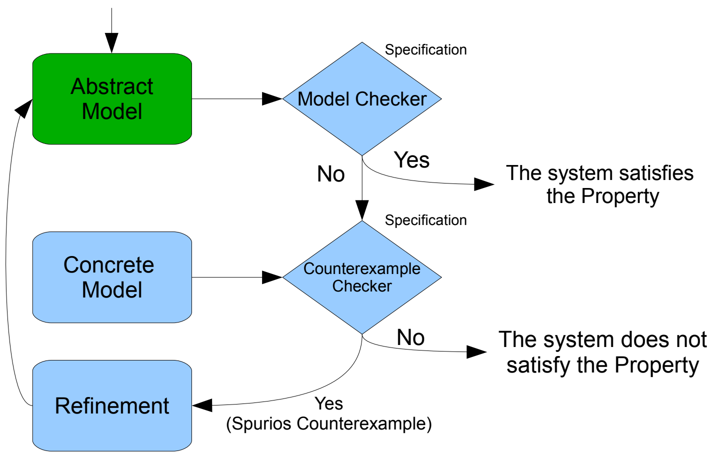

Figure 3: CEGAR Loop

Therefore they provide a symbolic algorithm to determine whether an abstract counterexample is spurious. If the counterexample is not spurious, we report it to the user and loop is stopped. If the counterexample is spurious, the abstraction must be refined to eliminate it. Instead when the property is true in the abstract model is it true also in the concrete model, so the property holds. In Figure 3 is represented the CEGAR verification loop.

4 Three-valued logic abstraction refinement

We propose for SPES model checking technique, with abstraction and refinement of the system. Our work may be considered as a generalization of the CEGAR approach (Section 3.3.3) in the setting of abstractions: Three-valued Abstraction Refinement (TVAR) [CGLT09] is based on three valued logic and we are presently implementing it. The specifications to verify are expressed by means of μ-calculus, which is a temporal logic, and the algorithm that checks these specifications is used for symbolic (BDD-based) model-checking (Subsection 3.3.1). Therefore we combine the major techniques used to reduce the state space explosion problem.

4.1 Model Abstraction

Our model checker is based on the notion of model abstraction (Section 3.3.2). An abstraction is conservative with respect to the property evaluation if the results in the abstraction verification have the same result in the correspondent concrete system. Traditionally, abstraction techniques are designed to be conservative when the result is $true$, that is, the property is satisfied by the abstraction. Thus, if a property is verified in the abstract system, it holds for the concrete system. If, however, a formula does not hold for the abstract system this may be because too much information has been hidden, say the abstraction is too coarse. In fact in CEGAR, Figure 3, a counterexample obtained in the abstract system is concretised and stepwise replayed, on the concrete system. This either shows that the property is indeed violated in the concrete system or it identifies a transition present in the abstract state machine system but not in the concrete. In this latter case, the abstraction may be refined by eliminating the corresponding transition in the abstract state machine system.

In multi-valued logic a formula evaluates no longer to just true or false but to one of many truth values. In our work we use three-valued logic, so the values are $true$, $false$, and $don't\ know$. In three-valued model checking, abstractions are considered to be conservative for $true$ and $false$ evaluations. It means that only when we have a $don't\ know$ result, we have to refine our abstraction, in the other cases the problem is solved also for the concrete system.

In order to have such framework Kripke structures are extended to the three-valued logic by assigning to each proposition in each state the values $true$, $false$ or $don't\ know$ and the same to each transition which means there is a transition, there is not, or may be. Abstract systems are so no longer normal Kripke structure but Larsen's and Thomsen's Kripke modal transition systems [LT88].

Kripke modal transition systems A Modal Transition System (MTS) [LT88] is a specification in state-transition form, where *loose* transitions (that is, transitions that may or may not be present in the final implementation) are labelled as *may*-transitions, and *tight* transitions, which must be preserved in the final implementation, are labelled as *must*-transitions. Since a transition that must be present in the final implementation may be present as well, every *must*-transition is by definition a *may*-transition. This is a foundation for three-valued program analysis.

We give an example of MTS in Figure 4, a specification of a slot machine, where some behaviours of the final implementation are fixed (the *must*-transitions) and some are uncertain (the *may*-transitions). Kripke modal transition systems, are a generalization of MTS, and are used to represent the abstract model meanwhile for the concrete one are used common Kripke structures. Transitions in the Kripke modal transition systems which are labelled as *may*-transitions may or may not be present in the concrete Kripke structure; transitions which are labelled as *must*-transitions must be present in the concrete structure. This leads to three possible values for a transition: It is there for sure, it is not there for sure, or it may be there. The notions of *must* and *may* transitions determine an over and under-approximation of the system.

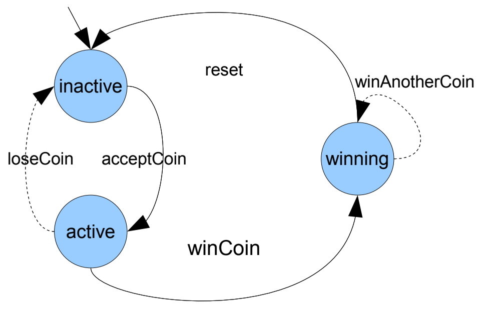

Figure 4: Kripke modal transition system example.

State abstraction We consider abstractions of Kripke structures induced by joining states to form abstract states, as is made in traditional logic systems. In this manner we can reduce the dimension of the Kripke structure joining concrete state.

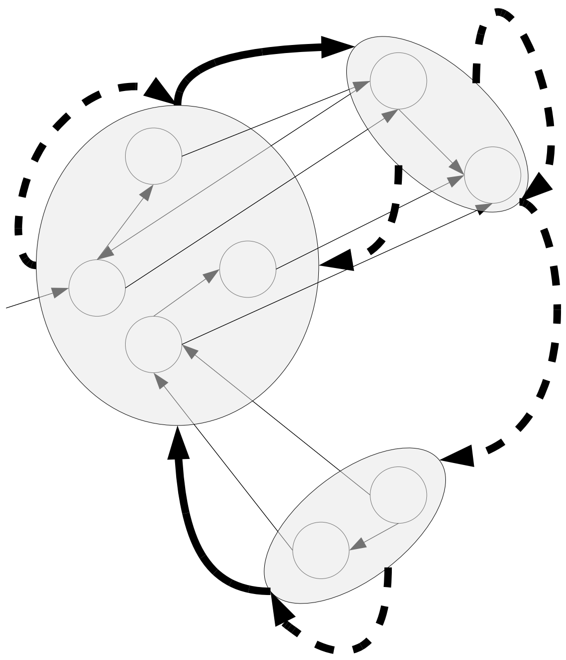

Figure 5: Over- and under- system approximation example.

In Figure 5 there is an example of a Kripke structure which shows state abstraction. The dotted transitions between abstract states represent *may*-transitions and the others are *must*-transitions. These transictions induce an over- and under- approximation of the system. The model checking procedure is made on this structure.

4.2 TVAR Model Checking

Model checking of the abstract system either yields *true* or *false*, which is called a definite result, or, the answer is *don't know* for an undefinite result. In the latter case, a *cause* for the indefinite result may be identified, which is a transition or state of the abstract system, for which it is only known that it *may* be available in the concrete system or, respectively, that it may satisfy some properties. Moreover, intuitively, this lack of knowledge is a reason for the indefinite result. The idea of identifying causes useful for refinement was first made precise in the context of three-valued CTL model checking by Grumberg and Shoham [SG07]. In case that model checking yields a definite result, i.e., *true* or *false*, this result also holds for the concrete system. If the model checking result is *don't know*, we moreover obtain a set of *causes*.

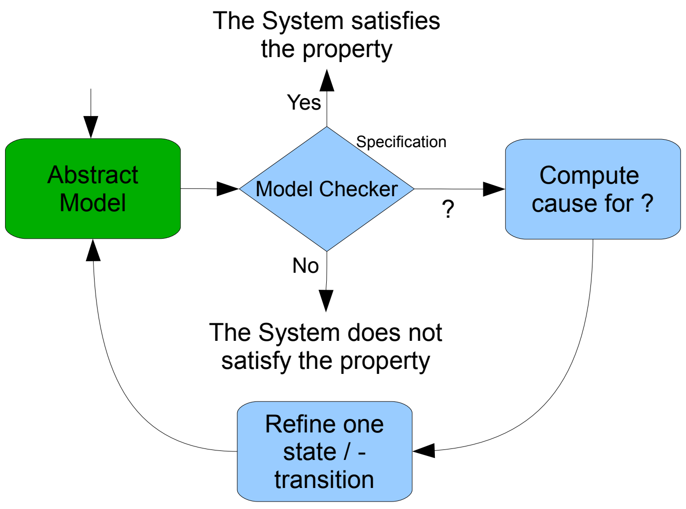
The System satisfies the property
Yes
Specification
Abstract Model
Model Checker
?
Compute cause for ?
No
The System does not satisfy the property
Refine one state / - transition

Figure 6: TVAR Loop

The resulting abstraction/refinement loop is shown in Figure 6 and is called Three-valued abstraction refinement loop. In comparison to CEGAR, TVAR has some advantages, thanks to the simultaneous over- and under approximation of the concrete system:

- TVAR allows verifying also existential properties that cannot be verified using CEGAR.
- While within CEGAR, a counterexample has to be replayed on the concrete system to obtain information for refinement, the cause for indefinite results can directly be obtained in the abstract system (in TVAR) and requires no further processing on the concrete system.

The price to pay for TVAR is that an underlying theorem prover/SMT solver, which is typically both employed in CEGAR and TVAR to reason on the existence of transitions, has to be more powerful in TVAR than in CEGAR.

4.3 Implementation

We have implemented the TVAR approach within the TVARC tool. It presently reads first-order system descriptions and our goal is to extend the input translating AutoFocus 3 model and adding support to C program (initially to C0). The properties are specified by μ-calculus and is implemented the TVAR loop, shown in the previous section. Many temporal logics, e.g., CTL, LTL, and CTL* can be easily translated into μ-calculus [Eme96], therefore we have a strong expressivity for the property definition.

We have implemented the TVAR loop based on the SMT solver Z3 [dMB08], whose rich API allow a convenient integration into our tool TVARC. The model checker is implemented on top of the JavaBDD-library. Instead of the usual state sets our model checker has to operate on elements of the three-valued logic introduced above. One component of our implementation maps three-valued structures to standard BDDs.

Results We have checked several existential properties of protocols, both with NuSMV [CCG+02] and TVARC, where we used integers of finite range for NuSMV and no restriction for TVARC. The general message of the experiments is that as long as NuSMV has no memory problems, it checks much faster than TVARC. While TVARC is typically slower than the highly-efficient model checker NuSMV, it uses far less memory. Thus, TVARC is able to verify systems in a reasonable amount of time systems that are far beyond the capabilities of NuSMV. Hence, this is a prove that TVAR will considerably extend the range of today's verification problems.

5 Future Work

Formal Verification Isabelle/HOL can be used to formulate and prove refinement in comparison to equivalence relations between different system specification resulting from different development phases and techniques. Interesting topics for future work are formal refinement/equivalence relations between Focus and AutoFocus 3 systems specifications as well as behavioural equivalence of AutoFocus 3 models and generated C0 code.

Model Checking In the Verisoft XT approach, as we explained in Section 2, the AutoFocus 3 models are verified implementing model checking techniques. We propose such an approach for SPES, by an integration with the model checkers SMV and TVARC. In such integration a future improvement would be the implementation of counterexample analysis, i.e. when the verification returns a negative result, the information provided by the counterexample might be related with the model and presented in a graphical way in the tool, in order to be useful for the debug phase. The TVAR tool is actually work in progress. We are implementing the support for programs written in the C language as input, providing at the beginning the support to C0 programs which are generated from AutoFocus 3 models. The following step is to support a direct conversion from an AutoFocus 3 model to a representation for TVARC. It will be interesting to work with a concrete tool in order to consider the performances in time and memory occupation, compared with the existing model checking tools (as for instance SMV).

Domain Analysis In order to provide a better understanding of our methodologies, we propose as future work a complete case study with concrete properties to be verified with the proposed analysis techniques. We are interested also to study the definition of classes of properties, which may be verified, referred to an application domain and/or pattern specifications suitable for SPES project. It will be also interesting to study the meaning (in terms of system reliability) for the failure or success results of the analysis techniques in selected classes of properties.

6 Conclusions

We have presented pervasive verification techniques for safety-critical embedded systems. In this document we have gave an overview of analysis techniques studied for FOCUS theory and its layers, and based on it a proposal for the SPES project. The artefacts of the different stages can be created through schematic transformations, which are, depending on the development state of the tools, automatic or at least tool supported. These artefacts are verified during each development phase: we apply both automatic and semi-automatic interactive verification techniques. Our proposal is mainly based on formal verification, applied to the AutoFOCUS and C0 layers of a FOCUS specification. We opt for methods in the framework of model checking and simulation. The advantages of such techniques fit with the criticality of embedded systems development.

We propose our own model checker based on Three-valued Abstraction Refinement (TVAR). Model checking is automatic, logics used can easily express many concurrency properties and in case of a violation provide counterexamples that can help in the debugging phase. The main disadvantage compared to other verification techniques is the state space dimension. In fact it may grow exponentially in the number of used variables or concurrent components, due to the exhaustive analysis performed. The common technique for reduce such problem are based on model abstraction. Our solution is in the same branch of CEGAR, but our work is based on three-valued logic system instead of common two valued logic, with notions of abstraction and refinement.

The proposed development methodologies make possible to perform formal verification on different levels of system abstraction and development, ranging from formalisation of system requirements to program code. The major verification techniques reported here are model checking and interactive theorem proving. We suggest to use model checking and simulation because they provide different features needed for different verification tasks and complementing each other. While model checking provides automatic verification for property notations of limited expressiveness like LTL, simulation allows to validate the model showing its execution. Besides the implementation of such techniques as future work we want to provide an easy specification way for the properties to verify and a counterexample analysis for improve the debug phase.

References

[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.

[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.

[Büc62] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Logic, Methodology and Philosophy of Science (Proc. 1960 Internat. Congr .)*, pages 1–11. Stanford Univ. Press, Stanford, Calif., 1962.

[Cam10] Alarico Campetelli. SPES 2020: Analysis Techniques: State of the Art in Industry and Research. Technical report, Technische Universität München, 2010.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[CCG$^+$02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *International Conference on Computer-Aided Verification (CAV'02, Proceedings)*, volume 2404 of *Lecture Notes in Computer Science*. Springer, July 2002.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement, 2000.

[CGLT09] Alarico Campetelli, Alexander Gruler, Martin Leucker, and Daniel Thoma. *Don't know* for multi-valued systems. In Zhiming Liu and Anders P. Ravn, editors, *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA'09)*, volume 5799, pages 289–305. Springer, 2009. to appear.

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

[CK96] Edmund M. Clarke and Robert P. Kurshan. Computer-aided verification. *IEEE Spectr.*, 33(6):61–67, 1996.

[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.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[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.

[FFH$^+$09] M. Feilkas, A. Fleischmann, F. Hölzl, C. Pfaller, K. Scheidemann, M. Spichkova, and D. Trachtenherz. A Top-Down Methodology for the Development of Automotive Software. Technical report, Technische Universität München, 2009.

[HHR09] Alexander Harhurin, Judith Hartmann, and Daniel Ratiu. Motivation and Formal Foundations of a Comprehensive Modeling Theory for Embedded Systems. Technical Report TUM-I0924, Technische Universität München, 2009.

[Höl09] F. Hölzl. The AutoFocus 3 C0 Code Generator. Technical Report TUM-I0918, Technische Universität München, 2009.

[LT88] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *LICS*, pages 203–210, 1988.

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

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

[SG07] Sharon Shoham and Orna Grumberg. A game-based framework for ctl counterexamples and 3-valued abstraction-refinement. *ACM Trans. Comput. Log.*, 9(1), 2007.

[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.

[Spi07] M. Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle.* PhD thesis, 2007.

[Tra09] David Trachtenherz. Ausführungssemantik von AutoFocus-Modellen: Isabelle/HOL-Formalisierung und Äquivalenzbeweis. Technical report, Institut für Informatik, Technische Universität München, January 2009.

[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.