The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems[⋆]

Marco Bozzano[1], Alessandro Cimatti[1], Joost-Pieter Katoen[2], Viet Yen Nguyen[2], Thomas Noll[2], and Marco Roveri[1]

[1] Fondazione Bruno Kessler, Trento, Italy
Tel.: +39 0461 314367; Fax: +39 0461 302040
bozzano@fbk.eu
[2] Software Modeling and Verification Group, RWTH Aachen University, Germany

Abstract. We report on a model-based approach to system-software co-engineering which is tailored to the specific characteristics of critical on-board systems for the aerospace domain. The approach is supported by a System-Level Integrated Modeling (SLIM) Language by which engineers are provided with convenient ways to describe nominal hardware and software operation, (probabilistic) faults and their propagation, error recovery, and degraded modes of operation.

Correctness properties, safety guarantees, and performance and dependability requirements are given using property patterns which act as parameterized "templates" to the engineers and thus offer a comprehensible and easy-to-use framework for requirement specification. Instantiated properties are checked on the SLIM specification using state-of-the-art formal analysis techniques such as bounded SAT-based and symbolic model checking, and probabilistic variants thereof. The precise nature of these techniques together with the formal SLIM semantics yield a trustworthy modeling and analysis framework for system and software engineers supporting, among others, automated derivation of dynamic (i.e., randomly timed) fault trees, FMEA tables, assessment of FDIR, and automated derivation of observability requirements.

1 Introduction

The design of modern space missions and systems poses fierce challenges. On the one hand, the involved systems are clearly critical, and huge amounts of money are at stake. On the other hand, the design involves the integration of a large number of heterogeneous requirements (e.g. functional correctness, dependability, observability, performance), for which different teams are responsible, and that often do not communicate in the early stages of the process.

In this paper, we describe an integrated, model-based methodology for system-software co-engineering, which is tailored to the specific characteristics of critical on-board systems for the space domain. The approach covers modeling,

⋆ Funded by ESA/ESTEC under Contract No. 21171/07/NL/JD.

functional correctness, and performance analysis. In terms of modeling, the approach is based on a System-Level Integrated Modeling (SLIM) language. The SLIM language is inspired by the well-known AADL [30] and provides engineers with convenient ways to describe nominal hardware and software operation, hybridity, (probabilistic) faults and their propagation, error recovery, and degraded modes of operation.

A fundamental feature of the approach is model extension: starting from a nominal model of the system, and a set of possible faults, the extension operator is able to generate a comprehensive description combining both the nominal and the faulty behaviours of the model. The SLIM language also allows for a comprehensive representation of partial observability, necessary to describe the actual sensing capabilities at the disposal of an on-line monitoring system.

The SLIM language allows to describe discrete dynamics, real time, and continuous dynamics, both in a qualitative and in a probabilistic fashion. A formal semantics allows to precisely characterize the complete set of nominal and non-nominal behaviours of the model, and opens up the possibility to apply a wealth of formal verification techniques for various forms of analysis. These include symbolic model checking for functional verification and formal requirements analysis, FTA and FMEA, testability, and performance analysis.

The activity described in this paper is inspired by the COMPASS project[1] (Correctness, Modeling, and Performance of Aerospace Systems). The project is in response to an invitation to tender by the European Space Agency. The methodology described in this work is made practical by a comprehensive toolset, called the COMPASS toolset, based on state of the art tools in verification, such as NuSMV [26], FSAP [19], Sigref [31], and MRMC [25]. The toolset and the methodology are currently under industrial evaluation and will be applied to several case studies by a major industrial developer of aerospace systems.

The paper is structured as follows. In Section 2, we describe the features of the SLIM language. In Section 3, we discuss how the various analyses can be reduced to (qualitative and quantitative) problems in formal verification. In Section 4, we present the structure of the COMPASS toolset. Finally, in Section 5 we draw some conclusions and outline directions for future work.

2 The Modeling Language

The System-Level Integrated Modeling (SLIM) language [7] has been designed in order to provide a cohesive and uniform approach to model heterogeneous systems, consisting of software (e.g., processes and threads) and hardware (e.g., processors and buses) components, and their interactions. Furthermore, it has been designed with the following essential features in mind.

– Modeling both the system's nominal and non-nominal behavior. To this aim, SLIM provides primitives to describe software and hardware faults, error propagation (that is, turning fault occurrences into failure events), sporadic

[1] `http://compass.informatik.rwth-aachen.de`

```
device Battery
  features
    empty: out event port;
    voltage: out data port real;
end Battery;

device implementation Battery.Imp
  subcomponents
    energy: data continuous initially 100.0;
  modes
    charged: initial mode
             while energy' = -0.01 and energy >= 20;
    depleted: mode
              while energy' = -0.015;
  transitions
    charged  -[when energy >= 15
               then voltage := f(energy)]-> charged;
    charged  -[empty when energy<20]->      depleted;
    depleted -[then voltage := f(energy)]-> depleted;
end Battery.Imp;
```

Fig. 1. Specification of a Battery Component

(transient) and permanent faults, and degraded modes of operation (by mapping failures from architectural to service level).
- Modeling (partial) observability and observability requirements. These notions are essential to deal with diagnosability and Fault Detection, Isolation and Recovery (FDIR) analyses.
- Specifying timed and hybrid behavior. In particular, in order to analyze continuous physical systems such as mechanical and hydraulics, the SLIM language supports continuous real-valued variables with (linear) time-dependent dynamics.
- Modeling probabilistic and quantitative aspects, such as probabilistic faults and performability measures.

The characteristics listed above make SLIM an ideal language to specify and reason about the following system properties: functional correctness, in particular in case of degraded hardware operation; safety and dependability; diagnosability and FDIR; system performance and performability.

2.1 Specifying Nominal Behavior

A SLIM model is hierarchically organized into *components*, distinguished into software (processes, threads, data), hardware (processors, memories, devices, buses), and composite components. Components are defined by their *type* (specifying the functional interfaces as seen by the environment) and their *implementation*(representing the internal structure). The implementation part contains:

```
system Power
  features
    voltage: out data port real;
end Power;

system implementation Power.Imp
  subcomponents
    batt1: device Battery.Imp in modes (primary);
    batt2: device Battery.Imp in modes (backup);
  connections
    data port batt1.voltage -> voltage
      in modes (primary);
    data port batt2.voltage -> voltage
      in modes (backup);
  modes
    primary: initial mode;
    backup: mode;
  transitions
    primary -[batt1.empty]-> backup;
    backup  -[batt2.empty]-> primary;
end Power.Imp;
```

Fig. 2. The Complete Power System

the structure of the component as an assembly of subcomponents; the interaction through (event and data) port connections; the (physical) binding at runtime; the operational modes as an abstraction of the concrete component behavior, possibly representing different system configurations and connection topologies, with mode transitions which are spontaneous or triggered by events arriving at the ports; the timing and hybrid behavior of the component. The overall specification can be organized into *packages* to support modularity.

To give a more concrete idea, Fig. 1 shows an example specification of a simple battery device. Its type interface features two ports: an outgoing event port `empty` which indicates that the battery is about to become discharged, and an outgoing data port `voltage` which makes its current voltage level accessible to the environment.

The corresponding component implementation specifies the battery to be initially in the `charged` mode with an `energy` level of 100 (%). This level is continuously decreased by 1% per time unit (in Fig. 1, `energy'` denotes the first derivative of `energy`) until a threshold value of 20% is reached, upon which the battery changes to the `depleted` mode. This mode transition triggers the `empty` output event, and the loss rate of energy is increased to 1.5%. Moreover, the `voltage` value is regularly computed from the `energy` level (the corresponding function, `f`, is not detailed here) and automatically made accessible to the environment via the corresponding outgoing data port.

The next specification, presented in Fig. 2, shows the usage of the battery component in the context of a redundant power system. It contains two instances

```
error model BatteryFailure
  features
    normal: initial state;
    dead:   error   state;
end BatteryFailure;

error model implementation BatteryFailure.Imp
  events
    fault: error event occurrence poisson 0.001;
  transitions
    normal -[fault]-> dead;
end BatteryFailure.Imp;
```

Fig. 3. An Error Model

of the battery device, being respectively active in the `primary` and the `backup` mode. The mode switch that initiates reconfiguration is triggered by an `empty` event arriving from the battery that is currently active. Moreover the `voltage` information of the active battery is forwarded via an outgoing data port.

2.2 Specifying Faulty Behavior

Nominal component specifications can be extended by *error models* to support safety and dependability analyses. For the sake of modularity, nominal specifications, error specifications, and their mutual association are separated from each other.

Again, an error model is defined by its type, its implementation, and its effect. An error model *type* defines an interface in terms of error states and (incoming and outgoing) error propagations. Error states are employed to represent the current configuration of the component with respect to the occurrence of errors. Error propagations are used to exchange error information between components. An error model *implementation* provides the structural details of the error model. It is defined by a (probabilistic) machine over the error states declared in the error model type. Transitions between states can be triggered by error events, **`reset`** events, and error propagations. Error events are internal to the component; they reflect changes of the error state caused by local faults and repair operations, and they can be annotated with occurrence distributions to model probabilistic error behavior. Moreover, **`reset`** events can be sent from the nominal model to the error model of the same component, trying to repair a fault which has occurred. Outgoing error propagations report an error state to other components. If their error states are affected, the other components will have a corresponding incoming propagation. An error effect is specified by expressions that overload the nominal assignments when the error occurs. Fig. 3 presents a simple error model for the battery device. It introduces a probabilistic error event, `fault`, which is assumed to occur once every 1000 time units on average.

```
system PowerSystem
  features
    voltage: out data port real;
    alarm: out data port bool initially false observable;
end PowerSystem;

system implementation PowerSystem.Imp
  subcomponents
    pow: system Power.Imp;
  connections
    data port pow.voltage -> voltage;
  modes
    normal: initial mode;
    critical: mode;
  transitions
    normal -[when voltage < 4.5 then alarm:=true]-> critical;
    critical -[when voltage > 5.5 then alarm:=false]-> normal;
end PowerSystem.Imp;
```

Fig. 4. The Complete Power System with an Alarm

Whenever this happens, the error model changes into the `dead` state, that could for instance be associated with `voltage` being constantly `0.0`.

2.3 Specifying Observability

In order to enable modeling of partial observability, the SLIM language allows the specifier to explicitly define the set of observables. For instance, in the battery example, we may assume that the output voltage of the power system is observable, whereas the internal status of the batteries and the occurrence of faults is not. Fig. 4 shows an example in which an alarm, modeled as an observable Boolean output signal, is raised whenever the voltage is lower than 4.5 volts. Once raised, the alarm is deactivated if the voltage increases to 5.5 volts.

2.4 Formal Semantics

To enable trustworthy modeling and analysis of systems, our SLIM language is equipped with a formal semantics (see [7]) that provides the interpretation of SLIM specifications in a precise and unambiguous manner. The semantics has been designed in such a way to conform to the environment described in [3], which encompasses different aspects of the development of reactive systems, from functional verification to safety analysis, dependability and diagnosability, within the framework of symbolic model checking.

The semantics of a nominal specification is defined on two levels, distinguishing between the local behavior of an active component and the interaction between active components via ports and connections. This interaction is highly dynamic as local transitions can cause subcomponents to become (in-)active, and can change the topology of event and data port connections. On the level of the formal model this means that both the activation status of components and their interconnection relation depend on the modes of the components.

When it comes to integrating faulty system behavior, first the association between nominal and error models has to be specified. In the example above, e.g., one would connect (every instance of) the `Battery` device to the `BatteryFailure` error model. The occurrence of an error event, or a propagation in an error model implementation, indicates a (local, respectively global) fault, and generally causes the transition to a new error state. *Failure effects* can be attached to error states in order to specify the impact of a fault to the nominal behavior of that component. Every such effect is defined by a list of assignments to the component's data elements that overrides the nominal transition effects in the presence of an error. In the case of the battery example, one could reset the `voltage` level to zero while being in error state `dead`.

The actual integration of the nominal and the error model, the so-called *(fault) model extension*, works similarly to the procedure described in [8]. It takes the nominal model and enriches it by the error model specification, thus producing an integrated model which represents both the nominal and the failure behavior. Informally, this model is obtained as follows. Its modes are pairs of nominal modes and error model states. The set of event ports is obtained by adding the error propagations to the original event ports, in order to represent the exchange of error information via propagations as event communication. Correspondingly, the set of event port connections has to be extended by propagation port connections. Finally, in the mode transition relation of the integrated model, all possible interleavings and interactions between the nominal and the error model have to be considered.

2.5 Comparison with AADL

The SLIM language covers a significant subset of AADL. Many features of AADL have been omitted (such as properties, extensions, prototypes, and flow specifications), and the set of available component categories has been reduced. Also some "mixed" concepts (such as **`event data`** ports or **`in out`** ports) have been omitted to simplify the implementation. There are, however, some extensions that have been introduced in our language to support the description of dynamic system behavior.

– Initialization values for data ports and data components have been added.
– To support mode history, **`initial`** and **`activation`** modes are distinguished. This allows to express that after a re-activation of a component due to a system reconfiguration, the component should resume its operation in the state in which it had previously been deactivated.
– Explicit binding relations between subcomponents (**`stored in`**, **`running on`**, **`accesses`**) have been introduced.
– To support the specification of timed and hybrid behavior, mode invariants (**`while`**), transition guards (**`when`**) and transition effects (**`then`**) have been added (similarly to the AADL Behavior Annex).

From the semantical perspective, as a difference with AADL, which supports asynchronous communication via event queues, the SLIM language is based on (possibly multi-way) *synchronous* event communication.

3 Analyzing System Specifications

In this section we discuss the main analysis capabilities of the COMPASS approach. The available functionalities are summarized by the use case diagram in Fig. 5.

3.1 Property Specification and Validation

Formal properties are increasingly being used to describe the qualitative and the quantitative requirements of electronic designs. These properties are used both for verification and as a means to describe the requirements for a system before it is built. The use of a formal language to state formal properties is a first and substantial step towards a high quality specification, as it makes subtle questions explicit that otherwise might be hidden in the ambiguity of natural language.

Within the COMPASS project, we use temporal logic properties to describe both the qualitative and the quantitative properties the system under analysis has to satisfy. Linear Temporal Logic (LTL) [28] and Computational Tree Logic [13] are used to express qualitative properties. Probabilistic Computation Tree Logic (PCTL) [21] and Continuous Stochastic Logic (CSL) [2] are used to express quantitative properties. The definition of properties from non expert users can be facilitated by the use of property patterns [17].

The COMPASS approach supports property validation, to check correctness and completeness of a set of properties [27]. First, it allows to check for *logical*

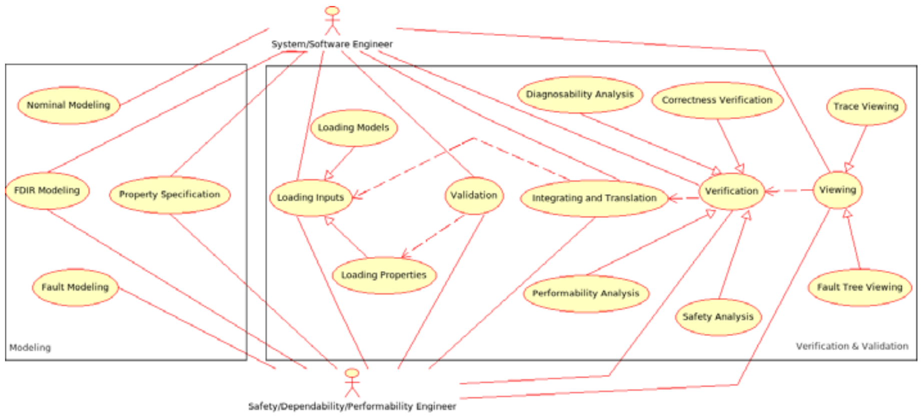

Fig. 5. Functionalities of the COMPASS approach

consistency. Logical consistency can be intuitively defined as “freedom from contradictions”: in fact it is possible that two properties mandate mutually incompatible behaviors. Consistency checking of temporal properties can be carried out by dedicated formal verification algorithms [11].

Second, it is possible to check the set of properties is *strict enough* to rule out unwanted behavior and that it is *not too strict* to disallow for certain desirable behavior. Checking that the properties are not too strict amounts to verifying whether a set of conditions (also called a scenario) is possible, given the constraints imposed by the considered set of properties. If the scenario is possible, we obtain a behavior trace compatible with both the properties and the constraint describing the scenarios. Otherwise, we obtain a subset of the considered set of properties that prevents the scenario to happen. Checking that the properties are strict enough to rule out unwanted behavior amounts to verifying whether an expected property (describing the desired behaviors) is implied by the considered set of properties. This check is similar in spirit to model checking [15], with the considered set of properties playing the role of the model. When the property is not implied by the specification, a counterexample, witnessing the violation of the property, is produced.

3.2 Verification of Functional Properties

A SLIM model can be evaluated using model checking techniques, in order to guarantee that it satisfies the required functional properties. To this aim, the model can be translated into a Labeled Transition System (LTS) and exhaustively analyzed by the model checker to check whether the properties hold. If a property does not hold, a counterexample trace can be generated to show an execution trace of the model that violates the property. To cope with the state explosion problem, advanced techniques can be applied, in particular symbolic techniques based on Binary Decision Diagrams (BDD) [9] and SAT-based Bounded Model Checking [4,5,22,18] (BMC). Verification can also benefit from advanced techniques for compiling temporal properties into a symbolic LTS [12].

In order to deal with the timed and hybrid domain (i.e., SLIM models containing integers and reals), standard symbolic model checking techniques cannot be applied. The most noticeable approach is Counterexample Based Abstraction Refinement [14] (CEGAR). Here, a property is verified in an abstraction of the original model. If verification is not conclusive, the abstraction can be automatically refined, based on analysis of the trace generated by the model checker, and the verification process is iterated. Advanced techniques for computing and refining the abstraction include techniques based on the emerging technology of Satisfiability Modulo Theory (SMT) [10]. Similar techniques can also be exploited in BMC. All these techniques have been incorporated into the NuSMV [26] model checker.

3.3 Verification of Safety/Dependability Aspects

The COMPASS methodology can be used to produce artifacts and support activities that are specific of safety assessment, such as techniques for hazard analysis.

The use of formal techniques for such activities is relatively new. The COMPASS methodology relies on the seminal work carried out within the ESACS[2] (Enhanced Safety Assessment for Complex Systems) and ISAAC[3] (Improvement of Safety Activities on Aeronautical Complex systems) projects, two European-Union-sponsored projects involving various research centers and industries from the avionics sector, and that resulted in the FSAP tool[19]. As advocated in [8], an essential step of the methodology is the decoupling between the nominal behavior and the faulty behavior of the system, that is realized by means of the model-extension step (cf. Section 2.4).

The COMPASS methodology supports two of the most popular hazard analysis techniques, that is, Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis (FTA). FMEA uses an inductive approach; it starts by considering the initiating causes of a given hazard, and traces them forward to the corresponding safety consequences. FTA, on the other hand, is a deductive technique; it starts by considering an unintended behavior of the system at hand, and traces it, in a backward reasoning fashion, to the corresponding causes. The COMPASS methodology can automatically generate (dynamic) fault trees [16,24], given an extended model and a property representing the hazard. Furthermore, (dynamic) FMEA tables can be automatically generated, given a set of failure modes (more in general, a set of fault configurations, which may include combinations of different faults) and a set of properties. Finally, it is possible to compute a criticality measure, which combines probability of occurrence and severity of the consequences.

3.4 Diagnosability Analysis

The COMPASS toolset support diagnosability analysis and FDIR (Fault Detection, Isolation and Recovery). These analyses are based on the notion of *observables* in the input model. In particular, fault detection analysis checks whether an observation can be considered a *fault detection means* for a given fault, that is, every occurrence of the fault eventually causes the observable to be true. All such observables are reported as possible detection means. Fault isolation analysis generates fault isolation measures, namely, for each of the observables, it generates a fault tree that contains the minimal explanations that are compatible with the observable being true (the fault tree contains one cut set consisting of a single fault, in case of perfect isolation). Finally, fault recovery verifies whether a user-defined recoverability property is satisfied. The COMPASS toolset can also check whether a system is diagnosable with respect to a diagnosability property, and synthesize a set of observables that ensure diagnosability.

3.5 Quantitative Analyses

To guarantee the required performance, a SLIM model can be evaluated using probabilistic model checking techniques [2]. Prior to this, the user has to specify

[2] `http://www.esacs.org`

[3] `http://www.cert.fr/isaac`

the formal performance requirements through PCTL or CSL properties: e.g. the system under degradation always has to recover within 40 time units with a probability of 0.98; or, that in the long run, the system will be down with a probability of 0.005. To check whether the SLIM model meets these requirements, it has to be transformed into its underlying Markov chain through probabilistic information captured by the `occurrences` definitions in the error models. The Markov Reward Model Checker [23] (MRMC) can then be used to evaluate whether the Markov chain meets the expressed performance requirements.

The same probabilistic model checking techniques are used for computing the probability of the top-level event in fault trees. They can be extended to computing probabilities for dynamic fault trees [6]. Akin to checking the correctness of FDIR measures, we use the same probabilistic techniques to evaluate FDIR performance. For example, in addition to checking whether a fault is detected or not, we compute the probability of detection; in case of fault recovery, we compute the probability that the system will recover from a fault.

Finally, it is possible to analyze the timing behaviour of a SLIM model, like for example whether the system will correctly reset a valve between 20 and 30 minutes. Clock invariants, constraints and resets expressed in the SLIM models are used for this. Drafting the transformations from these timing constructs to the underlying formal model, timed automata [1], is still work in progress.

4 Tool Support

The activities described in the previous sections are supported by an integrated platform, which incorporates extensions of existing tools in a uniform environment. Verification and validation functionalities of the toolset are based on symbolic model checking techniques. In particular, the tool set builds upon the NuSMV [26] symbolic model checker, the MRMC [25] probabilistic model checker, and the RAT [29] requirements analysis tool. The architecture of the tool set is shown in Fig. 6.

The toolset takes as input a model written in the SLIM language, and a set of property patterns [17,20]. It generates several artifacts as output, among them: traces resulting either from simulation of the SLIM specification or as counterexample for properties not satisfied by the specification; (probabilistic) Fault Trees and FMEA tables; diagnosability and performability measures.

In order to perform all the verification activities, the SLIM high-level specification is parsed and an internal representation of the input files and a symbol table are constructed. Depending on the specific verification task to be run, different transformations of the input files are then possible, and realized by the building blocks shown in Fig. 6. The ModelExtension block takes care of performing model extension, when required. It generates as output another SLIM model with probabilistic annotations (if any) that represent the faulty system. The Slim2SMV translator is used to translate a SLIM specification into a semantically-equivalent SMV file, which can be used for all NuSMV-based analyses, and to produce separate probabilistic information (if any). The safety analysis activities are performed by FSAP[19], which has been integrated within NuSMV. The SMV file

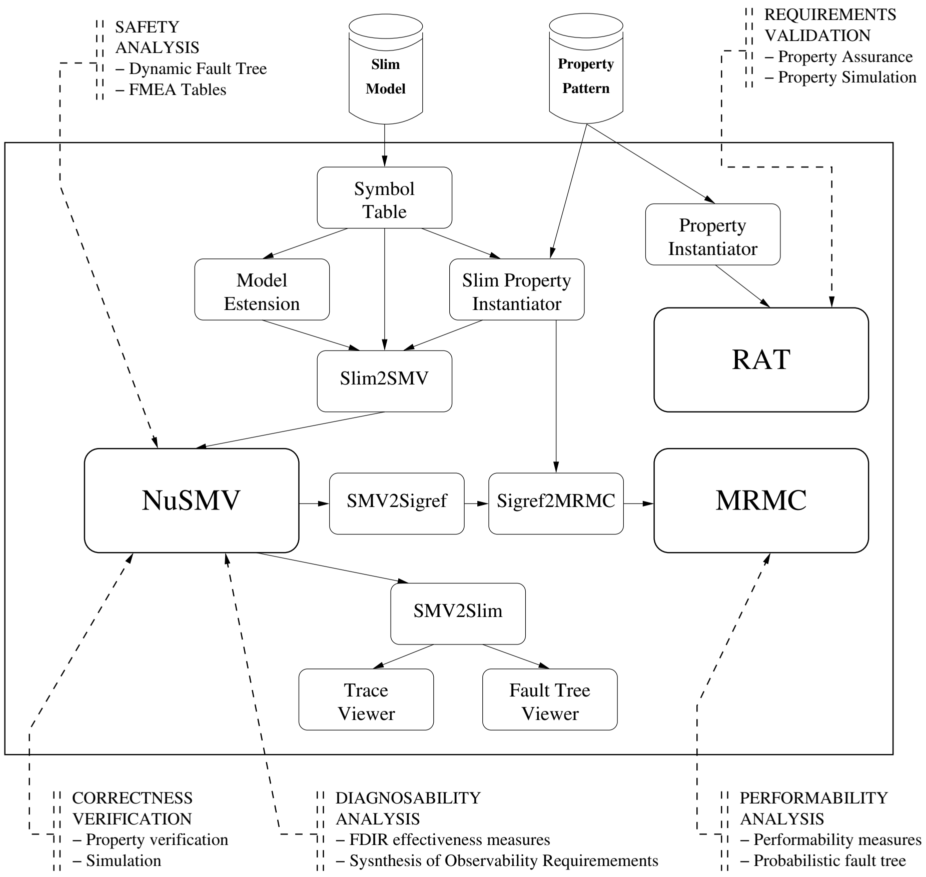

Fig. 6. Architecture of the COMPASS Platform

and the probabilistic information are used by the SMV2Sigref and Sigref2MRMC blocks, that collaborate to transform an SMV file into an equivalent input file for MRMC (the latter also contains probabilistic information), which can be used for all MRMC-based activities. Property patterns are used to create formal properties [17,20]. These properties are processed either by the Slim Property Instantiator, and then converted into SMV or MRMC format, or by the Property Instantiator, that transforms them into RAT format for requirements validation. Finally, the block SMV2Slim converts the results of the analyses back from the internal tools' format into SLIM format, which can be processed by the visualizers, namely graphical fault tree and trace viewers.

5 Conclusions and Future Work

In this paper, we presented a comprehensive, end to end methodology for the design of complex systems. The approach covers all possible user queries in a unique methodology, and it is formally well founded. It includes in a unique, clear formal framework, a number of analyses, and has a full-fledged support by the integration of several state of the art verification tools.

An industrial evaluation of the methodology on realistic case studies is currently ongoing within the COMPASS project. This will provide substantial insights on the applicability of the proposed methodology and the effectiveness of the tool chain. Of particular interest is the verification of reactive systems modeling continuous dynamics. In the future, we plan to systematically investigate the combination of symbolic model checking techniques for the effective construction of the state space to scale up quantitative and probabilistic analyses.

Acknowledgments

We wish to acknowledge and give special thanks to Benedikt Brütsch, Roberto Cavada, Christian Dehnert, Friedrich Gretz and Andrei Tchaltsev, that assisted us by programming a great part of the developed toolset and actively participated to the fruitful discussions that came along the way when we stumbled upon technical issues.

References

1. Audemard, G., Cimatti, A., Kornilowicz, A., Sebastiani, R.: Bounded Model Checking for Timed Systems. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529. Springer, Heidelberg (2002)
2. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. IEEE TSE 29(6), 524–541 (2003)
3. Bertoli, P., Bozzano, M., Cimatti, A.: A Symbolic Model Checking Framework for Safety Analysis, Diagnosis, and Synthesis. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS (LNAI), vol. 4428, pp. 1–18. Springer, Heidelberg (2007)
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
5. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods in Comp. Sc. 2(5) (2006)
6. Boudali, H., Crouzen, P., Stoelinga, M.: Dynamic fault tree analysis using input/output interactive Markov chains. In: DSN, pp. 708–717. IEEE, Los Alamitos (2007)
7. Bozzano, M., Cimatti, A., Nguyen, V.Y., Noll, T., Katoen, J.P., Roveri, M.: Codesign of Dependable Systems: A Component-Based Modeling Language. In: Proc. MEMOCODE 2009 (2009)
8. Bozzano, M., Villafiorita, A.: The FSAP/NuSMV-SA Safety Analysis Platform. International Journal on Software Tools for Technology Transfer 9(1), 5–24 (2007)
9. Bryant, R.: Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
10. Cavada, R., Cimatti, A., Franzén, A., Kalyanasundaram, K., Roveri, M., Shyamasundar, R.K.: Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In: Proc. FMCAD, pp. 69–76. IEEE Computer Society, Los Alamitos (2007)

11. Cimatti, A., Roveri, M., Schuppan, V., Tonetta, S.: Boolean abstraction for temporal logic satisfiability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 532–546. Springer, Heidelberg (2007)
12. Cimatti, A., Roveri, M., Tonetta, S.: Symbolic Compilation of PSL. IEEE Trans. on CAD of Integrated Circuits and Systems 27(10), 1737–1750 (2008)
13. Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8(2), 244–263 (1986)
14. Clarke, E., Grumberg, O., Jha, S., Lua, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. JACM, 752–794 (2003)
15. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2000)
16. Dugan, J., Bavuso, S., Boyd, M.: Dynamic fault-tree models for fault-tolerant computer systems. IEEE Transactions on Reliability 41(3), 363–377 (1992)
17. Dwyer, M., Avrunin, G., Corbett, J.: Patterns in property specifications for finite-state verification. In: Proc. ICSE, pp. 411–420. IEEE, Los Alamitos (1999)
18. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science 89(4) (2003)
19. The FSAP/NuSMV-SA platform, `http://sra.itc.it/tools/FSAP`
20. Grunske, L.: Specification patterns for probabilistic quality properties. In: ICSE 2008: Proceedings of the 30th international conference on Software engineering, pp. 31–40. ACM, New York (2008)
21. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
22. Heljanko, K., Junttila, T.A., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 98–111. Springer, Heidelberg (2005)
23. Katoen, J.-P., Khattri, M., Zapreev, I.: A Markov reward model checker. In: QEST, pp. 243–244. IEEE CS, Los Alamitos (2005)
24. Manian, R., Dugan, J., Coppit, D., Sullivan, K.: Combining Various Solution Techniques for Dynamic Fault Tree Analysis of Computer Systems. In: Proc. High-Assurance Systems Engineering Symposium (HASE 1998), pp. 21–28. IEEE Computer Society Press, Los Alamitos (1998)
25. The MRMC model checker, `http://wwwhome.cs.utwente.nl/~zapreevis/mrmc/`
26. The NuSMV model checker, `http://nusmv.itc.it`
27. Pill, I., Semprini, S., Cavada, R., Roveri, M., Bloem, R., Cimatti, A.: Formal analysis of hardware requirements. In: Proc. DAC, pp. 821–826. ACM, New York (2006)
28. Pnueli, A.: A temporal logic of concurrent programs. Th. Comp. Sc. 13, 45–60 (1981)
29. RAT: Requirements Analysis Tool, `http://rat.itc.it`
30. Architecture Analysis and Design Language (AADL) V2. SAE Draft Standard AS5506 V2, International Society of Automotive Engineers (March 2008)
31. Sigref — A Symbolic Bisimulation Tool, `http://sigref.gforge.avacs.org/`