An Organizing System to Perform and Enable Verification and Diagnosis Activities

Vincent LEILDE[1], Vincent RIBAUD[2], Philippe DHAUSSY[1]

[1] Lab-STICC, team MOCS, ENSTA Bretagne, rue François Verny, Brest
e-mail: firstname.lastname@ensta-bretagne.fr
[2] Lab-STICC, team MOCS, Université de Brest, avenue Le Gorgeu, Brest
e-mail: ribaud@univ-brest.fr

Abstract. Model-checkers increasing performance allows engineers to apply model-checking for the verification of real-life system but little attention has been paid to the methodology of model-checking. Verification “in the large” suffers of two practical problems: the verifier has to deal with many verification objects that have to be carefully managed and often re-verified; it is often difficult to judge whether the formalized problem statement is an adequate reflection of the actual problem. An organizing system - an intentionally arranged collection of resources and the interactions they support – makes easier the management of verification objects and supports reasoning interactions that facilitates diagnosis decisions. We discuss the design of such an organizing system, we show a straightforward implementation used within our research team and we present the possible ameliorations of the organizing system.

Keywords: verification, model-checking, diagnosis, organizing system

1 Introduction

System verification is used to establish that the design or product under consideration possesses certain properties. Formal verification has been advocated as a way forward to address verification tasks of complex embedded systems.Formal methods, within the field of computer science, is the formal treatment of problems related to the analysis of designs, but “it does not yet generally offer what its name seems to suggests, viz. methods for the application of formal techniques [1].”

Our research work is underlined by the observation that verification “in the large” causes a proliferation of interrelated models and verification sessions “that must be carefully managed in order to control the overall verification process [1].” This paper is concerned with practical ways to structure and control the verification and diagnosis activities. The main formal technique discussed in this paper is verification by model-checking. “Model checking is a formal verification technique which allows for desired behavioral properties of a given system to be verified on the basis of a suitable model of the system through systematic inspection of all states of the model [2].” Model-based verification techniques (including model-checking) are based on models describing the possible system behavior in a precise and unambiguous manner. “It turns out that – prior to any form of verification – the accurate modelling of systems often leads to the discovery of incompleteness, ambiguities, and inconsistencies in informal system specifications [3].”

Model-checking walks through different phases within an iterative process [3]:

Modelling phase: model the system under consideration using the model description language of the model checker at hand; as a first check and quick assessment of the model perform some simulations; formalize the property to be checked using the specification language.

Running phase: run the tool to check the validity of the property in the model.

Analysis phase: if the property is satisfied, then check next property (if any); if the property is violated, then analyze generated counterexample by simulation and refine the model, design, or property.

In addition to these steps, the entire verification should be planned, administered, and organized.

The applicability of model-checking to large systems suffers of two practical problems. A verification session refines the model or the design, and because properties are verified one by one, previously verified properties need or need not to be verified again, depending on the refinement performed. Moreover, if the model-checker runs out of memory, some divide-and-conquer techniques should be employed in order to reduce the model and try again. These techniques exploit regularities in the structure of the models or of the verification process itself that are difficult to understand and their performance may vary considerably.

A second practical problem arises from the difficulty to judge whether the formalized problem statement (model + properties) is an adequate reflection of the actual problem. This is also known as the validation problem or the problem of the validity of the formal model. If the verifier suspects the validity of a property, the property needs to be re-formalized and it starts the whole verification again. If the verifier suspects the validity of the design, the verification process restarts after an improvement of the design. It is admitted that the complexity of the involved system, as well as the lack of precision of the informal specification of the system’s functionality, makes it hard to answer the validation problem satisfactorily [1], [3].

Both problems require a more organized verification method (although organized should be an indispensable attribute of a method). Organizing creates or supports capabilities by intentionally imposing order and structure. In this paper, we will apply the concepts of an organizing system promoted by [4]: “an Organizing System is an intentionally arranged collection of resources and the interactions they support." As an attempt to solve the problems mentioned above, we designed and built a prototype of an Organizing System for the support of verification and diagnosis activities. In section 2, we precise the issues of the management of verification cycles; we introduce a general theory of diagnosis, used for solving the error interpretation and we presents some design decisions for our Organizing System. Section 3 deepens different aspects of an Organizing System: knowledge management and ontologies, technical aspects of the tiers of the organizing system, and its place in the whole verification toolset. Section 4 relates our work with previous work, and Section 5 concludes.

2 Organizing System for Verification and Diagnosis activities

2.1 Managing the Verification Trajectories

There are basically three possible outcomes of a verification run: the specified property is either valid in the given model or not, or the model is faced with the state space explosion problem (it turns out to be too large to fit within the computer memory).
If a state is encountered that violates the property under consideration, the model checker provides a counterexample that describes an execution path that leads from the initial system state to a state that violates the property being verified. It is indisputable that the verification results obtained using a verification tool should always be reproducible [5]. Tool support is required and we will present in Fig. 3 the objects that are significant and tool-managed during the verification phases. The specification-design-modelling-verification cycles are presented in Fig. 1. Baier and Katoen book [3] is used as a reference terminology in the paper.

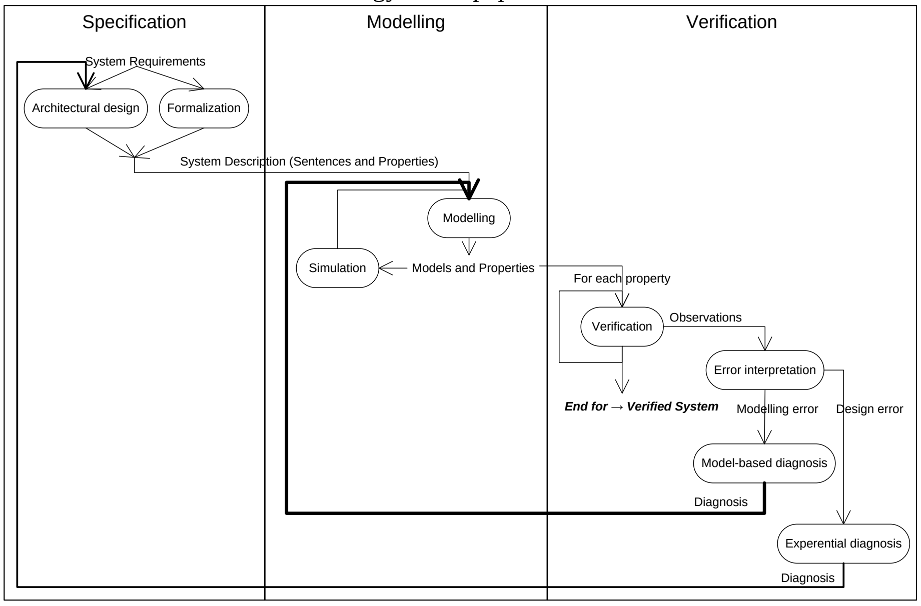

Fig. 1. The specification-design-modelling-verification process and its cycles

Modelling.
Model checking inputs are a model of the system under consideration and a formal characterization of the property to be checked. Models are mostly expressed using finite-state automata, consisting of a finite set of states and a set of transitions. To allow verification, properties are described using a property specification language, relying generally on temporal logic that allows people to describe properties in a precise and unambiguous manner. As mentioned above, it may be a serious problem (known as the validation problem) to judge whether the formalized problem statement (model + properties) is an adequate description of the actual verification problem [3].

Verification: running the model-checker.
Current model checkers provide the user with various options and directives to optimize and tune the functionality and performance of the verification run. Subsequently, the actual model checking takes place. Basically, model-checking explores all possible system states to check the validity of the property under consideration.
Whenever a property is not valid, it may have different causes. A modelling error means that the model does not reflect the design of the system. After a correction of the model, verification has to be restarted with the improved model. This corresponds to the bolded cycle in Fig. 1. When there is no undue discrepancy between the design and its model, then either a design error has been exposed, or a property error has taken place (this case is not considered in the paper). In case of a design error, the verification is concluded with a negative result, and the design (together with its model) has to be improved. This corresponds to the situation where the designer proceeds with iterative refinements and is depicted with a normal plain cycle in Fig. 1.

Interpreting the error(s).

From the debugging viewpoint, the main advantage of model checking is the production of counterexamples demonstrating that a system does not satisfy a specification. Extracting the essence of an error from even a detailed source-level trace of a failing run requires a great deal of human-effort [6] and a lot of research work focus on counterexample processing to produce an error explanation. Hence, correcting the error(s) starts with an error diagnosis, and more precisely, using the fault, error, and failure nomenclature of [7], it starts with a failure diagnosis. Failure diagnosis is the process of identifying the fault that has led to an observed failure of a system or its constituent components.

Model-based diagnosis.

The technique we use for failure diagnosis is a model-based diagnosis (also called reasoning from first principles) based on a theory of diagnosis established by [8], [9], [10]. A diagnosis is as a set of assumptions about a system component's abnormal behavior such that observations of one component's misbehavior are consistent with the assumptions that all the other components are acting correctly [10]. The computational problem is to determine all possible diagnoses for a given faulty system. The representation of the knowledge of the problem domain should achieve the desired coverage and quality of diagnoses while remaining computationally tractable [11].

Verification organization.

Whether the verification trajectory is incorporated in an adaptive design strategy or focused on modelling-and-verifying cycles, the entire model-checking process should be well organized, well structured, and well planned. According to [3], different model descriptions are made describing different parts of the system, various versions of the verification models are established, and plenty of verification parameters and results are available. Our proposal is to use an organizing system in order to manage a practical model-checking process and to allow the reproduction of the experiments carried out by the engineers.

It is normal to organize our world, but doing so systematically is key and the subject of The Discipline of Organizing (TDO) approach promoted by Robert J. Glushko and al. [4]. The authors made the observation that library and information science, informatics, computer science and other fields focus on the characteristic types of resources and collections that define those disciplines. In contrast, TDO complements the focus on specific resource and collection types with a framework that views organizing systems as existing in a multi-dimensional design space in which we can consider many types of resources at the same time and see the relationships among them. The framework is based on an assessment of what is being organized, why, how much, when and by what means. It leads to an Organizing System defined as “an intentionally arranged collection of resources and the interactions they support [4].”

To sum up the problem statement of this section, managing the verification trajectories is an indispensable support for using model checkers “in the large”. Moreover the proliferation of verification resources and the variety of possible interactions with them requires an Organizing System. “The concept of the Organizing System highlights the design dimensions and decisions that collectively determine the extent and nature of resource organization and the capabilities of the processes that compare, combine, transform and interact with the organized resources [4].”

In the next section, we will address the domain of diagnosis and some of its issues.

2.2 Theories of Diagnosis

A variety of failure diagnosis techniques drawing from diverse areas of computing and mathematics such as artificial intelligence, machine learning, statistics, stochastic modelling, Bayesian inference, rule-based inference, information theory, and graph theory have been studied in the literature [12]. “From a modelling perspective, there are methods that require accurate process models, semi-quantitative models, or qualitative models. At the other end of the spectrum, there are methods that do not assume any form of model information and rely only on historic process data [13].” Venkatasubramanian has broadly classified fault diagnosis methods into three general categories and reviewed them in three parts: quantitative model-based methods [13], qualitative model-based methods [14], and process history based methods [15].

In the model-based diagnosis, often referred as diagnosis from first principles, one is given a description of a system, together with an observation of the system's behavior which conflicts with the way the system is meant to behave. “The diagnostic problem is to determine those components of the system which, when assumed to be functioning abnormally, will explain the discrepancy between the observed and correct system behavior [10].” Under the process-history based methods, only the availability of large amount of historical process data is needed. The structure or the design of the real system being diagnosed is only weakly represented, or not at all. “Successful diagnoses stem from the codified experience of the human expert being modeled, rather than from what is often referred to as "deep" knowledge of the system being diagnosed [10].” Without denying the importance of the experiential approaches, we use a theory of diagnosis from fist principles based on Reiter’s work [10] as a general theory of diagnosis. The theory is presented in Fig. 2, with a summary after the figure.

Definition 2.1. A system is a pair (SD, COMPONENTS) where:

(1) so, the system description, is a set of first-order sentences;

(2) COMPONENTS, the system components, is a finite set of constants.

In all intended applications, the system description will mention a distinguished unary predicate AB(*), interpreted to mean "abnormal."

Diagnostic settings involve observations. Without observations, we cannot determine whether something is wrong and whether a diagnosis is called for.

Definition 2.2. An observation of a system is a finite set of first-order sentences. We shall write (SD, COMPONENTS, OBS) for a system (SD, COMPONENTS) with observation OBS.

Suppose we have determined that a system (SD, $\{c_1, \ldots, c_n\}$) is faulty, by which we mean informally that we have made an observation OBS which conflicts with what the system description predicts should happen if all its components were behaving correctly. Now $\{\neg AB(c_1),..., \neg AB(c_n)\}$ represents the assumption that all system components are behaving correctly, so that so SD U $\{\neg AB(c_1),. . . , \neg AB(c_n)\}$ represents the system behavior on the assumption that all its components are working properly. Hence the fact that the observation OBS conflicts with what the system should do were all its components behaving correctly can be formalized by:

SD U $\{\neg AB(c_1) ..... \neg AB(c_n) \}$ U OBS (1) is inconsistent.

Intuitively, a diagnosis is a conjecture that certain of the components are faulty (ABnormal) and the rest normal. The problem is to specify which components we conjecture to be faulty. Now our objective is to explain the inconsistency (1), an inconsistency which stems from the assumptions $\neg AB(c_1)$ .... ,$\neg AB(c_n)$, i.e. that all components are behaving correctly. The natural way to explain this inconsistency is to retract enough of the assumptions $\neg AB(c_1)$ .... ,$\neg AB(c_n)$, so as to restore consistency to (1). But we should not be overzealous and exhibit a diagnosis where all components are faulty.

Definition 2.3. A diagnosis for (SD, COMPONENTS, OBS) is a minimal set Δ est inclus dans COMPONENTS such that SD U OBS U $\{AB(c) \mid c$ appartient à $\Delta\}$ U $\{\neg AB(c) \mid c$ appartient à COMPONENTS $- \Delta\}$ is consistent.

In other words, a diagnosis is determined by a smallest set of components with the following property: The assumption that each of these components is faulty (ABnormal), together with the assumption that all other components are behaving correctly ($\neg$ABnormal), is consistent with the system description and the observation.

Fig. 2. Reiter, R. (1987). A theory of diagnosis from first principles. Artificial intelligence, 32(1), 57-95.

One begins with a description of a system, including desired properties and the structure of the system's interacting components. Whatever one's choice of representation, the description will specify how that system normally behaves on the assumption that all its components are functioning correctly. We need a diagnosis if we have available an observation of the system's actual behavior and if this observation is logically inconsistent with the way the system is meant to behave. Intuitively, a diagnosis determines system components which, when assumed to be functioning abnormally, will explain the discrepancy between the observed and correct system behavior [10]. There may be several competing explanations (diagnoses) for the same faulty system, the computational problem, then, is to determine all possible diagnoses.

2.3 Design Decisions

Explicitly or by default, establishing an Organizing System (OS) requires many decisions. These decisions are deeply intertwined, but it is easier to introduce them as if they were independent. In [4], authors introduce five groups of design decisions.

What is being organized?

System models, verification runs and diagnosis are our primary source of interest. There are all made of digital resources, but we can make a distinction between primary resources (such as [parts of] models, properties, verification runs, and counterexamples) and description resources that describe the primary resources and/or their relationships. Verification benchmarks (such as the BEEM — BEnchmarks for Explicit Model checkers [16]) provide valuable inputs and need to be organized in collections. Any OS user can also organize her/his own verification endeavors in collections, sub-collections of models, properties, runs and results.

Why it is being organized?

OS users are modelling and verification engineers, working alone or in teams, who need "to deal with the data explosion of the modelling phase and the versioned product space explosion of the verification phase [1]." The OS gathers and organizes quantitative and qualitative information to support knowledge creation and automated diagnosis reasoning. OS users share knowledge without being constrained to espouse a given formalism. OS users need to navigate efficiently through the resources space. The OS supports a reverification procedure to make sure that errors found in the model do not invalidate previous verification runs.

How much is it being organized?

The simplest OS can be a software configuration management system controlling the release and change of each digital resource, leaving the burden of organization outside of the OS. At the opposite of the spectrum, the OS can be a full ontology where any relationship between any piece of information is carefully defined and controlled, allowing many reasoning possibilities. For our point of view, the OS manages essentially documents (models, results, traces). Each document organizes its knowledge structure and content, according to its document type, and the engineer writes and reads information according to this structure. The reification of the underlying knowledge structure for reasoning purposes is done automatically by the OS.

When is it being organized?

The OS is intended to assist the engineer in his/her daily modelling and verification tasks, hence resources are organized continuously. However, the OS should offer an ingestion feature that helps to enter inputs into the OS. Ingest feature provides the services and functions to accept complex verification endeavors or benchmark collections and prepares the contents for storage and management within the OS. Conversely, an access feature provides the services and functions that support users in determining the existence, description, location and availability of information stored in the OS, and allowing users to extract information in a parametrized manner.

How or by whom, or by what computational processes, is it being organized?

Although a single verification engineer will benefit of the OS use, the OS is intended to support teamwork and to share knowledge about models and verification endeavors. As mentioned above, automated processes should extract as much knowledge as possible from the documents internal structure and from the collections organization. As a collaborative teamwork, organization is performed in a distributed, bottom-up manner.

3 Inside the Organizing System

3.1 Knowledge Management and Ontologies

In [17], the authors state that knowledge is an enterprise's most important assets and define the basic activities of knowledge management: identification, acquisition, development, dissemination, use, and preservation of the enterprise's knowledge. They advocate a corporate or organizational memory (OM) at the core of a learning organization, supporting sharing and reuse of individual and corporate knowledge and lessons learned [17]. The concept of an organizing system intended to knowledge management (KM) is a modern reincarnation of the organizational memory and we can benefit from the results gained in this research area. Knowledge acquisition and maintenance pose a serious challenge for organizational memories and [17] recommend adhering to the following principles: - exploit easily available information sources ; - forgo a complete formalization of knowledge; - use automatic knowledge-acquisition tools; - encourage user feedback and suggestions for improvements; - check the consistency of newly suggested knowledge.

An organizational memory or a KM organizing system relies substantially on existing information sources, which constitute the first tier of its architecture, called the object level in [17] and the storage tier in [4]. This level or tier is characterized by a variety of sources, heterogeneous with respect to several dimensions concerning form and content properties. An organizational memory or an information organizing system offers presentation facilities in the access tier, called the application-specific level in [17] and the presentation tier in [4]. This level or tier performs the mapping from the application-specific information needs to these heterogeneous object-level sources via a uniform access and utilization method on the basis of a logic-based, knowledge-rich level, a middle tier called the knowledge description level in [17] and the logic tier in [4]. The knowledge-rich level has the central role of a shared language to connect people to people, people to information, and information to information [18], and the level includes ontologies as a core enabler. As major knowledge-based KM applications, ontologies are used for the following three general purposes [18]: to support knowledge visualization; to support knowledge search, retrieval, and personalization; to serve as the basis for information gathering and integration.

Fig. 3 represents the main concepts and relationships of our ontology. A system is referred to by several propositional objects: system requirements (sentences and formalized properties), the system model (and its components), observations generally made about verification runs that are organized within verification endeavors. According to Reiter's theory presented in Fig. 2, a diagnosis is a conjecture that certain of the components are faulty (Abnormal) and the rest normal, stemming from an observation inconsistent with the system descriptions. All information on a particular verification run is not detailed here, and include, among others, checked properties, run outcomes, model-checker options and statistics.

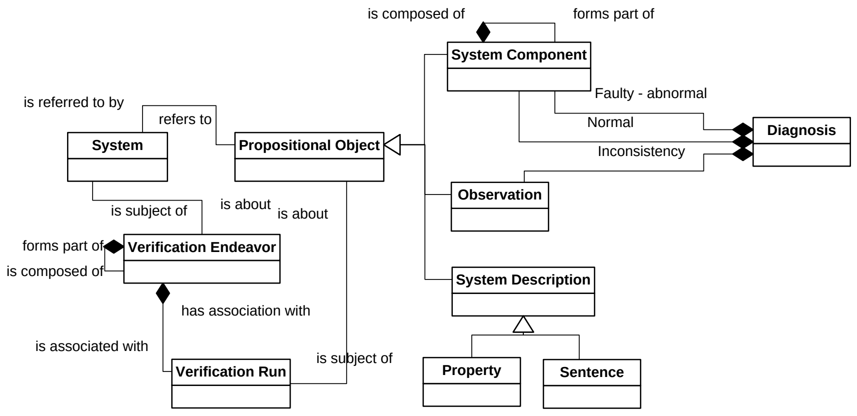

Fig. 3. Main elements of the ontological level.

3.2 Technical Aspects of the Tiers of the Organizing System

Modern applications separate the storage of data, the business logic or functions that use the data, and the user interface or presentation components through which users or other applications interact with the data. For each tier, we kept the design as simple as possible, relying on straightforward and widely-used solutions.

The storage tier.

As mentioned in the introduction, a practical difficulty of using model checkers "in the large" is the management of all (generated) data during the verification endeavors.

A disciplined recording of information on the different models during the verification phase becomes even more indispensable when errors are found, because, once the erroneous models have been corrected and re-verified, all models that have been verified previously and which are affected by the error should be re-verified as well [1]. We propose to use a Software Configuration Management (SCM) system to control the versioned artifacts produced during modelling and verification phases.

We do not impose any arrangement to the verification engineers. Hence, a verification endeavor is associated with a directory, with a total freedom to arrange endeavors and runs in a recursive manner. Using the combination of SCM feature and tools (e.g., the tool make) able to process automatically the building of software artifacts, each engineer organizes her/his endeavors in an arranged hierarchy or a rake of runs.

Complex objects such as set of properties or models decomposition are managed in the same manner, with a root directory and a freedom of organization. In order to ease the integration of each single object at the logical level, an XML description file stores information about the objects, a feature called a version description in a SCM system. The structure of the XML (its schema) is used by the software components (providing ingest and access features) to maintain an up-to-date ontological network in the logical level.

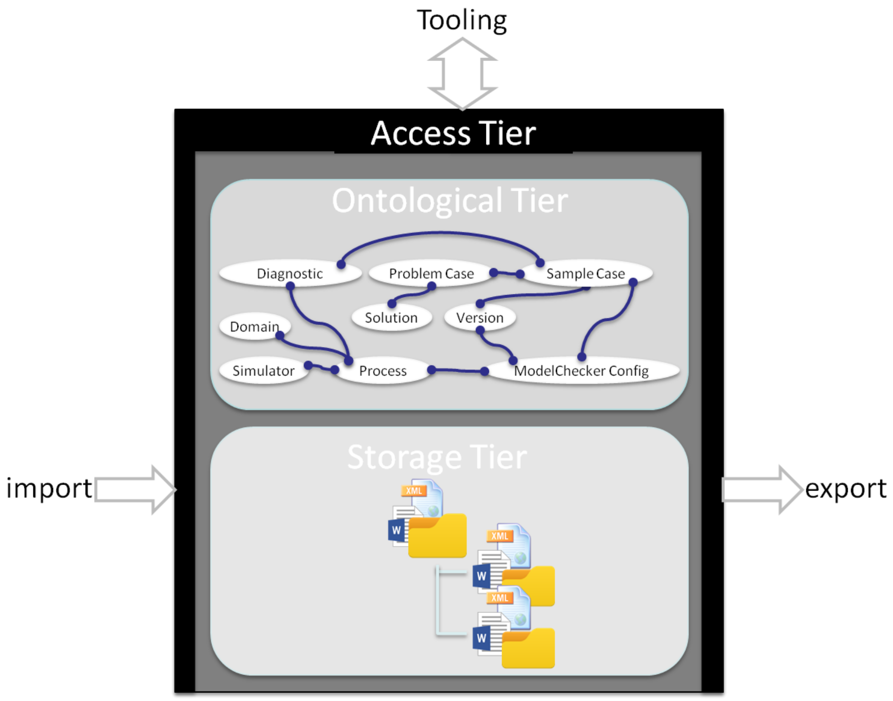

Fig. 4. The three tiers of the Organizing System

Avoiding the building of an information silo was also a concern. A silo is an insular management system that is unable to operate with any other systems. Files, directories, XML description, source version control ensure an access independent of any management system.

The access tier.

One of the main goals of an Organizing System is to support the design and implementation of the actions, functions or services that make use of the resources. In classical 3-tier architecture, the presentation tier is the tier in which users interact with an application. Typical user interactions are ingestions (importing new resources into the OS), searches, browsing, tagging and annotations, retrieval, information extraction. An Organizing System is also intended to interact with other applications and should provide information exchange features with or without semantic transformations. Because any of the user interactions mentioned above might be performed in a dedicated tool, in this perspective, the presentation layer is merely an access layer and its main concern is tool interoperability. Among possible solutions, we choose a pragmatic approach, called conceptual interoperability based on the concept of the federation of models [19] and its open source tooling (http://research.openflexo.org). The Openflexo tool set provides support for building conceptual views expanding upon existing models and tools; the modelling environment is divided into three responsibilities or spaces: an open and free conceptual modelling space, the designer interactions, projections in the technological or information spaces.

The ontological tier.

From a conceptual perspective, the ontological layer is divided into two parts. The semantic network of types, on one hand, consists of semantic types linked by types of semantic relations, equivalent to an entity-relationship model or an UML class diagram). The semantic network of objects, on the other hand, contains a node for each fine-grained object plus nodes for the composite objects, each of which is assigned to one or more semantic types and linked to other objects by semantic relations.

The ontological tier is evolving continuously over time, because new resources types and new resources can be added at any time. It is unrealistic to expect that all people and organizations developing information and knowledge application systems will use a common, shared ontology [20]. We have to live with different ontologies and there will be a need to reconcile these ontologies with a common upper ontology. Because we are accustomed to the CIDOC CRM ontology, a standardized structure (ISO 21127:2014) for describing the implicit and explicit concepts and relationships used in cultural heritage documentation, we use the CIDOC CRM as an upper ontology (http://www.cidoc-crm.org/official_release_cidoc.html), but we have to examine other candidates, such as SUMO [21] or DOLCE [22].

Technically, the ontological layer is stored in a TDB triple store; and we use the Apache Jena API to extract data from and write to RDF graphs (http://jena.apache.org/). There is an isomorphism between physical objects in the storage layer and semantic objects in the ontological layer; it is the responsibility of the access layer to maintain the isomorphism when items are added in the storage tier.

3.3 The Big Picture : place of the Organizing System in the Whole

Recall that our research work aims to contribute to the solving of two practical problems: to deal with many verification objects (that have to be carefully managed and often re-verified); and to facilitate the judgement of the validation problem (is the formalized problem statement an adequate reflection of the actual problem?).

The first problem requires essentially a methodology of verification and a support tool. Each verification engineer work process is made of slightly different activities using their own resources through different verification tools; hence each engineer defines her/his methodology or uses a given one. Our aim is to provide an integration framework for the tools and methodologies; this is precisely the goal of an Organizing System to arrange resources and to support interactions with. As a next step, the OS will be assessed using three fundamental criteria, understandability, reproducibility, and usefulness [23].

The second problem is part of a larger problem of computer-supported diagnosis that, as mentioned in Section 2.2, has been addressed within several solution spaces and a multitude of techniques. Our hypothesis is that the logical tier of the OS, knowledge-rich and ontology-based, serves as the basis for information gathering, information integration, knowledge creation and knowledge sharing. Thanks to the access layer, tools interoperability is made easier and tools collaboration provides the user with the required help. The OS acts as a backbone and we present in this section some experiments we already made and that we will add as plugins for computer-aided diagnosis: visualization, omniscient debugging, and case-based reasoning.

Visualization.

The formal model of the system-under-study (SUS), for instance concurrent state-machines, is used to exhaustively explore the SUS possible configurations. Then properties to be satisfied by the SUS have to be specified, and weaved with the SUS model. The weaving yields a Labeled Transition System (LTS) whose exploration permits to assert if a property is satisfied or not. If the property is violated the model-checker produces a counter-example trace. The analysis of large model exploration is almost impossible without tool support; visualization tools exploit the human ability to quickly understand complex visual patterns. A large number of visualization tools exists for studying traces; they exploit a wide range of diagram structures ranging from waveforms to large graph visualizations [24].

Omniscient debugging.

In case a model fails to satisfy a property, the model-checker offers counterexamples serving as indispensable debugging information. However, diagnosis is made difficult for several reasons: the trace conforms to a structure that is internal to the verification tool and hence hard to exploit, the trace yields low-level information, the trace size can be large. Omniscient debuggers, also known as back-in-time debuggers, record the whole history, or execution trace, of an execution of the debugged program. Omniscient debuggers make it possible to navigate backwards in time within an execution trace, drastically improving the debugging of complex applications [25].

Case-based reasoning.

The main limitation of model-based diagnosis is that it requires a model. What does mean a modelling error in a model-checking approach? It means that, upon studying the error it is discovered that the model does not reflect the design of the system and that implies a correction of the model [3]. Hence it means that the design is the subject of diagnosis, and that we need a correct model of the design (that we do not have, in essence) to apply model-based reasoning. Fortunately, we can use case-based reasoning (CBR) to find this correct model. In CBR, a reasoner remembers previous situations similar to the current one and uses them to solve the new problem. So, we need to describe the old cases (called Problem Cases) in the OS using a Problem Case Template (mainly the problem statement, the formalized properties, the “correct” model and implementations for different model-checkers). Reasoning on a new case (called Sample Case) suggests “a model of reasoning that incorporates problem solving, understanding, and learning and integrates all with memory processes [26].”

Experimentations.

Our team develops and maintains a model-checking tool kit. The SUS is described using the Fiacre language [27], which enables the specification of interacting behaviors and timing constraints through timed-automata. Our approach, called Context-aware Verification, focuses on the explicit modelling of the environment as one or more contexts. Interaction contexts are described with the Context Description Language (CDL). CDL enables also the specification of requirements through predicates and properties. The requirements are verified within the contexts that correspond to the environmental conditions in which they should be satisfied. All these developments are implemented in the OBP tool kit [28] and are freely available[1]. We designed a visualization front-end for the LTS and traces [29] and a trace query language called KriQL, featuring a blend of set filters and graph-based operations [30]. A work using pattern for relate Problems and Sample Cases is under submission [31].

4 Related work

The research work of Theo C. Ruys, from his PhD [32] to his recent tools [33] is the closest to ours, particularly for the first research problem addressed in this paper. The concept of managing the verification trajectory [1], by Ruys and Brinksma, has been a seminal paper for the understanding of the verification cycles and the need for a Software Configuration Management System for the verification “in the large” of real-life systems. We differ in scope because Ruys’s work is focused on the use of the SPIN model-checker while we are looking for an agnostic view of model-checking that implies an intermediate abstract (and ontological) layer between the verification engineer and her/his verification objects. We have common practical goals and we agree that the verification engineer uses a given model-checker (e.g. SPIN) and her/his proper management methodology and would not pay an added price to use the

[1] 1OBP Languages and Tool kit website: http://www.obpcdl.org

Organizing System, i.e. to describe and manipulate verification objects and results at an abstract level. Because any resources ingested in the Organizing System is going through the access layer, the only extra price to pay for the verification engineer is to describe her/his organization documents (e.g. the version descriptor used in [1]; containing a description of the files included in a particular version) in a schema (e.g. in a XML Schema definition) and to relate schema components (e.g. element and attribute declarations and complex and simple type definitions) with semantic types and/or semantic relations of the OS ontological layer. If the verifiers' schema components does not exist in the ontological layer, the verification engineer has to indicate whether semantic constructs they refine and missing components will be added in the ontology. Thanks to this mapping, the particular view of any verification engineer will be shared with the other users of the OS.

Hence, our work and Ruys's work address the same issues and rely on the same solution scheme. However, our approach has two main advantages: it supports any model-checking tool and method and enlarges the community of OS users; the ontological layer permits shared knowledge and reasoning over different model-checking verification experiences, working across boundaries.

The research work of the Divine team [34] was a second source of inspiration. Divines verifies models in multiple input formats and has excellent execution performances using a cluster of multi-core machines and partial order reduction techniques, breaking through the limits of the state space explosion problem. However, our work has been more influenced by side products of the Divine team, mainly issues related to the BEEM benchmark management [16] and the automation of the verification process [34]. Automation is necessary for practical applicability of formal verification. Pelanek states his work in such terms "given a system and a property, find a technique T and parameter values p such that T(p) can provide answer to the verification problem. This can be viewed as a verification meta-search problem [35]." He agrees for the need for classifications based on a model structure and also classifications based on features of state spaces, which relate to model-based or experiential diagnosis introduced in Section 2.2. Pelanek's work perspectives mention a long term goal intended to develop an automated 'verification manager', which would be able to learn from experience [35]. Our approach is more humble and pragmatic: to provide the user with the bigger possible set of knowledge about verification, including an ontological classification of the problem and the solution spaces. Thanks to the arranged knowledge within the Organizing System, the verification engineer can plug her/him plug tools to address the automated verification manager issue in her/his way

Alrajeh and al work is also a source of inspiration. They state that "Model checking and Inductive Logic Programming (ILP) can thus be seen as two complementary approaches with much to gain from their integration [36]." In their approach, counter-examples (false execution sequences) and witnesses (positive execution sequences) are the key data for this integration. Model checking provides ILP with a precise context for learning the most relevant hypotheses in the domain being studied, and ILP supplies model checking with an automatic method for learning corrections to models. In a broad perspective, the inductive learning of a concept focuses on examples showing how the concept is used whether the deductive learning explains a given concept and follows this explanation with examples. Hence, an ILP approach does not care too much on a semantic description of the problem and solution spaces whereas this point is fundamental in our Organizing System approach. However, Alrajeh and al. work give us the direction for integrating artificial intelligence and machine-learning techniques in the quest of solving our second research problem related to error interpretation and diagnosis.

5 Conclusion

Verification "in the large" suffers of two practical problems: the verifier has to deal with many verification objects that have to be carefully managed and often re-verified; it is often difficult to judge whether the formalized problem statement is an adequate reflection of the actual problem. We designed and built a prototype of an organizing system (OS) - an intentionally arranged collection of resources and the interactions they support – that makes easier the management of verification objects and supports reasoning interactions that facilitates diagnosis decisions.
Key points and driving issues of this research work are the usability of the OS, hosting a large variety of model-checking tools, techniques and methods; and the interoperability of the OS with external tools, providing the user with the freedom to use the proper approach to her/his problems.
However, we keep in mind that "any verification using model-based techniques is only as good as the model of the system. [3]." Hence, a particular attention to the validity of the problem formalization (known as the validation problem) will drive our future research efforts.

6 References

1. Ruys, T.C., Brinksma, E.: Managing the verification trajectory. International Journal on Software Tools for Technology Transfer (STTT). 4, 246–259 (2003).
2. Larsen, K.G., Pettersson, P., Yi, W.: Model-checking for real-time systems. In: International Symposium on Fundamentals of Computation Theory. pp. 62–88. Springer (1995).
3. Baier, C., Katoen, J.-P.: Principles of model checking. The MIT Press, Cambridge, Mass (2008).
4. Glushko, R.J. ed: The Discipline of Organizing. The MIT Press, Cambridge, MA (2013).
5. Holzmann, G.J.: The Theory and Practice of A Formal Method: NewCoRe. In: IFIP Congress (1). pp. 35–44 (1994).
6. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Model Checking Software. pp. 121–136. Springer (2003).
7. Avižienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. Dependable and Secure Computing, IEEE Transactions on. 1, 11–33 (2004).
8. DEKleer, J.: Local Methods for Localizing Faults in Electronic Circuits. (1976).

9. Genesereth, M.R.: The use of design descriptions in automated diagnosis. Artificial Intelligence. 24, 411–436 (1984).
10. Reiter, R.: A theory of diagnosis from first principles. Artificial intelligence. 32, 57–95 (1987).
11. Peischl, B., Wotawa, F.: Model-based diagnosis or reasoning from first principles. IEEE Intelligent Systems. 18, 32–37 (2003).
12. Kavulya, S.P., Joshi, K., Giandomenico, F.D., Narasimhan, P.: Failure Diagnosis of Complex Systems. In: Wolter, K., Avritzer, A., Vieira, M., and van Moorsel, A. (eds.) Resilience Assessment and Evaluation of Computing Systems. pp. 239–261. Springer Berlin Heidelberg, Berlin, Heidelberg (2012).
13. Venkatasubramanian, V., Rengaswamy, R., Yin, K., Kavuri, S.N: A review of process fault detection and diagnosis: Part I: Quantitative model-based methods. Computers & chemical engineering, 27, 293-311(2003).
14. Venkatasubramanian, V., Rengaswamy, R., Kavuri, S.N.: A review of process fault detection and diagnosis Part II: Qualitative models and search strategies. Computers and Chemical Engineering. 27, 313–326 (2003).
15. Venkatasubramanian, V., Rengaswamy, R., Kavuri, S.N., Yin, K.: A review of process fault detection and diagnosis: Part III: Process history based methods. Computers & chemical engineering. 27, 327–346 (2003).
16. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Model Checking Software. pp. 263–267. Springer (2007).
17. Abecker, A., Bernardi, A., Hinkelmann, K., Kühn, O., Sintek, M.: Toward a Technology for Organizational Memories. IEEE Intelligent Systems. 13, 40–48 (1998).
18. Abecker, A., van Elst, L.: Ontologies for knowledge management. In: Handbook on ontologies. pp. 713–734. Springer Berlin Heidelberg (2009).
19. Guychard, C., Guerin, S., Koudri, A., Beugnard, A., Dagnat, F.: Conceptual interoperability through Models Federation. In: Semantic Information Federation Community Workshop (2013).
20. Hameed, A., Preece, A., Sleeman, D.: Ontology Reconciliation. In: Staab, P.D.S. and Studer, P.D.R. (eds.) Handbook on Ontologies. pp. 231–250. Springer Berlin Heidelberg (2004).
21. Niles, I., Pease, A.: Towards a Standard Upper Ontology. In: Proceedings of the International Conference on Formal Ontology in Information Systems - Volume 2001. pp. 2–9. ACM, New York, NY, USA (2001).
22. Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., Schneider, L.: Sweetening Ontologies with DOLCE. In: Gómez-Pérez, A. and Benjamins, V.R. (eds.) Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web. pp. 166–181. Springer Berlin Heidelberg (2002).
23. Spackman, K.A., Reynoso, G.: Examining SNOMED from the perspective of formal ontological principles: Some preliminary analysis and observations. In: KR-MED. pp. 72–80 (2004).
24. Hamou-Lhadj, A., Lethbridge, T.C.: A Survey of Trace Exploration Tools and Techniques. In: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research. pp. 42–55. IBM Press, Markham, Ontario, Canada (2004).
25. Pothier, G., Tanter, É., Piquer, J.: Scalable omniscient debugging. ACM SIGPLAN Notices. 42, 535 (2007).
26. Kolodner, J.: Case-based reasoning. Kaufmann, San Mateo, Calif (1997).

27. Berthomieu, B., Bodeveix, J.-P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F., Vernadat, F.: Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. Presented at the ERTS 2008 January (2008).
28. Dhaussy, P., Boniol, F., Roger, J.-C., Leroux, L.: Improving Model Checking with Context Modelling. Adv. Soft. Eng. 2012, 9:9–9:9 (2012).
29. Boudaoud, S.R., Es-Salhi, K., Ribaud, V., Teodorov, C.: Relational and graph queries over a transition system. In: EUROCON 2015. , Salamanca, Spain (2015).
30. Es-Salhi, K., Boudaoud, S.R., Teodorov, C., Drey, Z., Ribaud, V.: KriQL: a query language for the diagnosis of transition systems. In: AVOCS'14. , Enschede, Netherlands (2014).
31. Leilde, V., Ribaud, V., Dhaussy, P. Organizing problem and sample cases for model-based diagnosis. Submitted in Second International Workshop on Patterns in Model Engineering co-located with MODELS'16, Saint-Malo, France (2016).
32. Ruijs, T.C.: Towards Effective Model Checking, http://doc.utwente.nl/36596/, (2001).
33. Ruys, T.C.: Unit Testing for SPIN: Runspin and Parsepan. In: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software. pp. 133–136. ACM, New York, NY, USA (2014).
34. Barnat, J., Brim, L., Havel, V., Havlíček, J., Kriho, J., Lenčo, M., Ročkai, P., Štill, V., Weiser, J.: DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: Sharygina, N. and Veith, H. (eds.) Computer Aided Verification. pp. 863–868. Springer Berlin Heidelberg (2013).
35. Pelánek, R.: Model classifications and automated verification. In: Formal Methods for Industrial Critical Systems. pp. 149–163. Springer (2007).
36. Alrajeh, D., Russo, A., Uchitel, S., Kramer, J.: Integrating Model Checking and Inductive Logic Programming. In: Muggleton, S.H., Tamaddoni-Nezhad, A., and Lisi, F.A. (eds.) Inductive Logic Programming. pp. 45–60. Springer Berlin Heidelberg (2011).