From EU Projects to a Family of Model Checkers
From Kandinsky to KandISTI

Maurice H. ter Beek, Stefania Gnesi, and Franco Mazzanti

Formal Methods && Tools lab (FM&&T)
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" (ISTI)
Consiglio Nazionale delle Ricerche (CNR)
Via G. Moruzzi 1, 56124 Pisa, Italy
{terbeek,gnesi,mazzanti}@isti.cnr.it

Abstract. We describe the development of the KandISTI family of model checkers from its origins nearly two decades ago until its very recent latest addition. Most progress was made, however, during two integrated European projects, AGILE and SENSORIA, in which our FM&&T lab participated under the scientific coordination of Martin Wirsing. Moreover, the very name of the family of model checkers is partly due to Martin Wirsing's passion for art and science.

1 Introduction

We have had the pleasure to work with Martin in two European projects, namely FP5-IP-IST-2001-32747 AGILE [2] and the FP6-IP-IST-016004 SENSORIA [57]. He coordinated both in an excellent manner.

AGILE created primitives for explicitly addressing mobility in architectural models. Therefore algebraic models based on graph transformation techniques were defined for the underlying processes to relate the reconfiguration of the coordination structure and the mobility of components across the distribution topology. Moreover, an extension of UML for mobility was developed to make the architectural primitives available to practitioners, together with tool support.

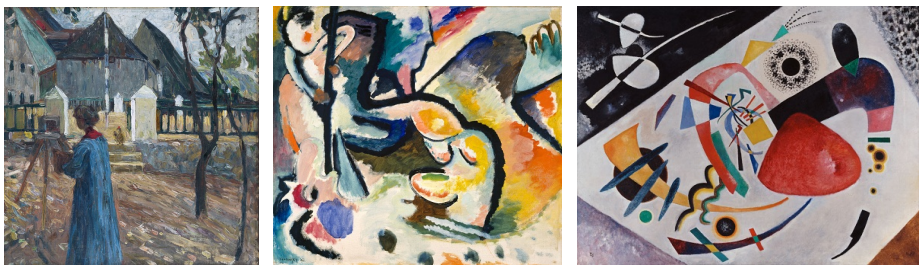
SENSORIA resolved problems from Service-Oriented Computing (SOC) by building novel theories, methods, and tools supporting the engineering of software systems for service-oriented overlay computers. The results include a comprehensive service ontology, new semantically well-defined modeling and programming primitives for services, new powerful mathematical analysis and verification techniques, tools for system behavior and quality of service properties, and novel model-based transformation and development techniques [57].

Based on our expertise, our involvement in AGILE was mainly to develop analysis techniques to support compositional verification of properties addressing the evolution of computation, coordination and distribution. In SENSORIA, instead, we developed a logical verification framework for the analysis of functional properties in SOC. In both projects, our work was strongly focused on the realization of a model-checking framework, as a result of which we now have a family of model checkers that we will describe in this paper.

This paper is organized as follows. In Sect. 2 we explain the name we gave to our family of model checkers, after which we briefly describe each family member in Sect. 3–6. We then sketch their overall structure in Sect. 7, after which we conclude the paper in Sect. 8.

2 From Kandinsky to KandISTI

In the beginning of 2008, one of the SENSORIA meetings included as social event a visit to the Lenbachhaus, a museum which preserves one of the richest collections of Wassily Kandinsky. At that time we were in the middle of the process of reshaping our family of model checkers by separating the specification language dependent details of the underlying *ground* computational model from its abstract representation in terms of a so-called Doubly-Labeled Transition System (L^2TS) [33], on which to carry out the analysis. While our ground computational models (state machines, process algebras) are already a simplified model of a real system, their correspondence with reality is still very immediate as they directly reflect the real system structure and behavior. Observing a model at this level, as explicitly allowed by our model-checking framework, is like exploring the real system which is being modeled. In some sense, our ground models are similar to the early paintings of Kandinsky (*e.g.* Fig. 1(a)) in which the correspondence of the painting to the reality is immediate. [1]



(a) Kallmünz - Gabriele Münter Painting II, 1903 (b) St. George III, 1911 (c) Red Spot II, 1921

Fig. 1. By Wassily Kandinsky (Städtische Galerie im Lenbachhaus, Munich, Germany)

In a very short time, however, Kandinsky's style of painting started to evolve into a more abstract style and his paintings started to no longer directly reflect reality in all its details. Instead, the painter chose to communicate just what he felt was relevant to him (*e.g.* Fig. 1(b)). Again, this is precisely what we intend

[1] The depicted thumbnails of Kandinsky paintings are among those observable on the official website of the Städtische Galerie im Lenbachhaus in Munich, Germany (*cf.* `http://www.lenbachhaus.de/collection/the-blue-rider/`), and are used here for non-commercial and strictly illustrative purposes.

to achieve in our family of model checkers, when we define *abstraction rules* which allow to represent the system as an L^2TS in which the labels on the states and edges directly represent just the abstract pieces of information we want to observe, to be able to express the properties we want to verify. A specific feature of our framework displays the model precisely at this abstraction level, even if at this level we are still able to find a correspondence between the abstract L^2TS and the underlying computational model, since each state and each edge can still be mapped back to a precise system state and system evolution.

In the last series of Kandinsky paintings, the disconnection between the observed reality and the represented images is almost complete (*e.g.* Fig. 1(c)). His paintings directly express just the author's feelings that the observation of reality stimulates. In our framework we have the possibility to apply to our abstract L^2TS a powerful minimization technique, which allows to observe in a graphical and very concise way the system behavior with respect to the abstract pieces of information we have selected to observe. In this way the resulting picture loses its direct connection with the underlying model (it is no longer possible to map a node to a single system state) and directly communicates most of the system properties regarding the observed aspects of the system. The intuition on the correctness of a system can be gained by just observing the representation of its abstract minimized behavior.

During the aforementioned visit to the Lenbachhaus, Martin Wirsing did not fail to notice the reminiscence of the various abstraction levels of our verification framework to the various approaches to painting through which Kandinsky's style has evolved, and we enjoyed together this wonderful matching. This visit inspired us to name our ISTI verification framework in a way that somehow reflects and honors Kandinsky's contribution to the art of painting, and this is why we have decided to name it KandISTI.

The development of the KandISTI family of model checkers is an ongoing effort [24, 40, 17]. The current versions of its family members are freely usable online via:

`http://fmt.isti.cnr.it/kandisti/`

On that page you will see the front-end of the family depicted in Fig. 2 and by clicking on one of its family members the specific tool will open.

In the next four sections, we briefly describe the different computational models underlying the model-checking tools of KandISTI, after which we will describe the unique logical verification environment in more detail in Sect. 7.

3 FMC: The Origin of Our On-the-Fly Model-Checking Approach

Experiments at ISTI with on-the-fly model checking began with the FMC model checker [39] for action-based CTL (ACTL) [32] extended with fixed-point operators. In FMC, a system is a hierarchical composition (net) of sequential automata (terms). Terms can be recursively defined using a simple process algebra which supports features coming from CCS, CSP and LOTOS [36]. Communication and synchronization among terms is achieved through synchronous operations over

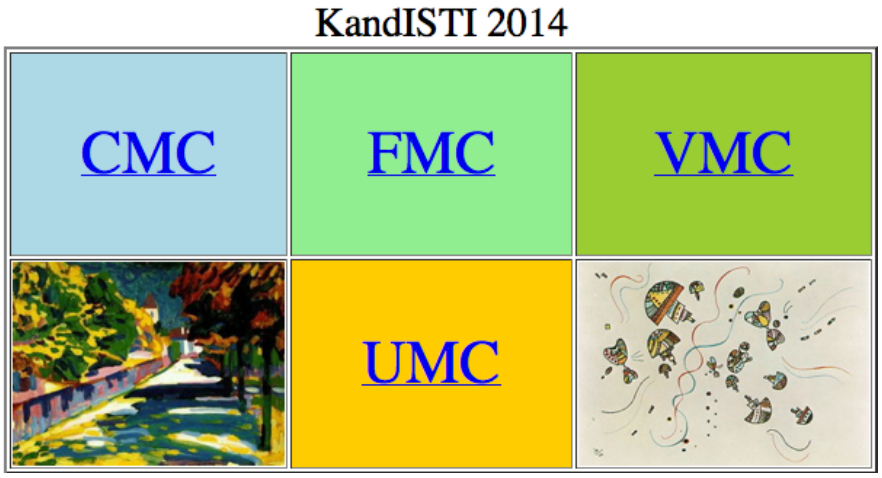


Fig. 2. The front-end of the KandISTI family of model checkers

channels. The parallel operator / *Channels* / defined in the syntax below allows the CCS synchronization between two participating networks, requires the CSP-like synchronization when the participating networks evolve with a communication action controlled by the specified list of Channels, and lets the participants proceed in interleaving when executing CSP actions not explicitly controlled. Moreover, all participants of a communication/synchronization must agree on the set of values exchanged during the operation.

All this allows to naturally model both binary client-server interactions and n-ary barrier-like synchronizations. Term definitions can be parametrized, and communication operations allow value passing. The only supported form of values are integer numbers, stand-alone identifiers can also be used as values and behave like special implementation defined integer constants.

Summarizing, the structure of the process algebra accepted by FMC is described by the following abstract syntax (where only the case in which term definitions and communication actions have precisely one parameter is depicted, but obviously their number can be arbitrary):

$$\begin{array}{l} \mathit{System} \ ::= \ [\,\mathit{Net}\,] \\ \mathit{Net} \ ::= \ T(\mathit{expr}) \ \mid \ \mathit{Net} \ / \ \mathit{Channels} \ / \ \mathit{Net} \ \mid \ \mathit{Net} \setminus \mathit{channel} \ \mid \ \mathit{Net}\,[\,\mathit{channel}/\mathit{channel}\,] \end{array}$$

where $[\,\mathit{Net}\,]$ denotes a *closed* system, *i.e.* a process that cannot evolve on actions that rely on input parameters ($\mathit{channel}(?\mathit{variable})$ as defined below); $T(\mathit{expr})$ is a process instantiation from the set of process declarations of the form $T(\mathit{variable}) \stackrel{\text{def}}{=} \mathit{Term}$; and *Channels* is a list of channel names. Next to the *parallel* operator / *Channels* / mentioned before, $\setminus$ *channel* and $[\,\mathit{channel}/\mathit{channel}\,]$ denote the classical operators of channel *restriction* and *renaming*, respectively.

The structure of *Term* definitions is described by the following abstract syntax:

$$
\begin{array}{l}
\mathit{Term} \ ::= \ \mathit{nil} \ \mid \ T(\mathit{expr}) \ \mid \ \mathit{Action}.\mathit{Term} \ \mid \ \mathit{Term} + \mathit{Term} \ \mid \ [\,\mathit{expr} \bowtie \mathit{expr}\,]\,\mathit{Term} \\
\mathit{Action} \ ::= \ \mathit{channel}(\mathit{arg}) \ \mid \ ?\mathit{channel}(\mathit{arg}) \ \mid \ !\mathit{channel}(\mathit{arg}) \\
\mathit{arg} \ ::= \ \mathit{expr} \ \mid \ ?\mathit{variable} \\
\mathit{expr} \ ::= \ \mathit{variable} \ \mid \ \mathit{integer} \ \mid \ \mathit{identifier} \ \mid \ \mathit{expr} \pm \mathit{expr}
\end{array}
$$

where $\bowtie \in \{<, \leq, =, \neq, \geq, >\}$ is a comparison operator and $\pm \in \{+, -, \times, \div\}$ is an arithmetic operation.

The basic idea underlying the design of FMC is that, given a system state, the validity of a formula on that state can be evaluated analyzing the transitions allowed in that state, and analyzing the validity of a subformula in only some of the next reachable states, recursively. In this way (depending on the formula) only a fragment of the overall state space might need to be generated and analyzed in order to produce the correct result. Such model-checking procedures are also called *local*, in order to distinguish them from those called *global*, in which the whole state space is explored to check the validity of a formula (*cf.* [27, 7]).

For the evaluation of a formula, in order to be able to partially deal also with infinite-state systems (potentially introduced by the presence of integer values), a so-called *bounded* model-checking approach is adopted (*cf.* [27, 7]). The evaluation is started by assuming a certain value as a maximum depth of the evaluation. If the evaluation of the formula reaches a result within the requested depth, then the result holds for the whole system; otherwise the maximum depth is increased and the evaluation is retried (preserving all useful partial results already found). This approach, initially introduced to address infinite state spaces, can turn out to be useful also for another reason: by setting a small initial maximum depth and a small automatic increment of this bound at each re-evaluation failure, once a result is finally found then we might also have a usable explanation for it. Note, however, that depending on the structure of the formula (*e.g.* requesting a check on all reachable states) and on the structure of the model (*e.g.* of a too big size[2]) no result might be returned by the tool when all the available resources (*e.g.* memory) are consumed.

The logic initially supported by FMC is an action-based branching-time logic inspired by ACTL and enriched with weak until operators, box and diamond operators and fixed-point operators. The fragment of this logic without fixed-point operators allows verifications with a complexity which is linear with respect to the size of the model and the size of the formula. With the integrations of the other tools of the family this logic has been over the time extended with the new features introduced for the support of state properties and data correlations among actions.

So far, FMC has been used mainly in didactic contexts for the experimentation of various modeling and verification techniques. Its main limit for heavier industrial use is the lack of support for more structured data types (*e.g.* lists, sets, maps, vectors).

[2] The current limit for an exhaustive verification is a statespace of millions of states.

4 UMC: Support for State/Event-Based Models and Logics

As an attempt to reduce the gap between software engineers and theoreticians, the very same model-checking approach that was adopted for FMC has subsequently been applied to a computational model directly inspired by UML statecharts (*cf.* `http://www.uml.org`). This prompted the switching to an action- *and* state-based logic, that would allow to express in a natural way not only properties of evolution steps (*i.e.* related to the executed actions) but also internal properties of states (*e.g.* related to the values of object attributes). The result of this process has been the UMC model checker and its associated UCTL logic [14].

The initial part of the design, development, and experimentation of the approach has been carried out in the context of the AGILE project. The purpose of the project was the development of an architectural approach in which mobility aspects could be modeled explicitly. The project proposed extensions of UML to support mobile and distributed system design, including linguistic extensions of the UML diagrammatic notations, extensions of the Unified Process and a prototype for simulating and analyzing the dynamic behavior of designs of mobile and distributed systems.

According to the UML paradigm, a dynamic system is seen as a set of evolving and communicating objects, where objects are class instances. The set of objects and classes which constitute a system can be described in UML by a structure diagram, while the dynamic behavior of the objects can be described by associating a statechart diagram to their classes. Each object of the system will therefore behave like a state machine; it will have a set of local attributes, an event pool collecting the events that need to be processed, and a current progress status. The progress status of a state machine is given by the set of currently active states of the statechart diagram.

In UMC a system is described as a set of communicating UML-like state machines. The structure of a state machine in UMC is defined by a Class declaration, which has the following general structure:

```
class <name> is
  Signals:
  <list of asynchronous signals managed by the class' objects>
  Operations:
  <list of synchronous call ops managed by the class' objects>
  Vars:
  <list of local vars belonging to the class' objects state>
  Behavior:
  <list of rules defining state evolutions of the class' objects>
end <name>
```

The **Behavior** part of a class definition describes the possible evolutions of the system. This part contains a list of transition rules which have the following generic form:

```
<Source> --> <Target> {<EventTrigger>[<Guard> ] /<Actions> }
```

Each rule intuitively states that when the system is in state *Source*, the specified *EventTrigger* is available and the *Guard* is satisfied, then all *Actions* of the transition are executed and the system state passes from *Source* to *Target*.

In UMC, the actual structure of the system is defined by a set of active object instantiations. A full UMC model is defined by a sequence of Class and Object declarations and by a final definition of a set of Abstraction rules. The overall behavior of a system is in fact formalized as an abstract L^2TS and the Abstraction rules allow to define what we want to see as labels of the states and edges of the L^2TS.

This approach to model the abstract system behavior as an L^2TS, showing only the essential information for the verification of system properties, proved to be a winning idea. Hence it was applied also to the other tools of the family, thus allowing the development of a common logical verification layer for our family of model checkers, which consequently became independent from the details of the particular specification language and computational model of the various tools.

The logic initially supported by UMC was just an extension of the logic supported by FMC with the possibility of using state predicates and pure CTL-like operators. As we will see in the next section over time this logic has been extended with the new features introduced for the support of parametric formulas allowing to express data correlations among actions.

The development and the experience gained with UMC has also helped in clarifying the overall purpose for the development of our verification framework. The main purpose of our tools is not just the final validation step of a completed architectural design, but rather a formal support during all steps of the incremental design phase (*i.e.* when ongoing designs are still likely to be incomplete and, with a high probability, contain mistakes). Indeed, the UMC framework has evolved having in mind the requirements of a system designer as end user: (s)he intends to take advantage of formal approaches to achieve an early validation of the system requirements and an early detection of design errors. Therefore, the main goals of the development of UMC have been:

1. The possibility to manually explore a system's evolutions and to generate a summary of its behavior in terms of minimal abstract traces.
2. The possibility to investigate abstract system properties by using a branching-time temporal logic supported by an on-the-fly model checker.
3. The possibility to obtain a clear explanation of the model-checking results in terms of possible evolutions of the selected computational model.

In AGILE, planes landing and taking off from airports and transporting other mobile objects, namely passengers, were considered as an example of mobile objects. In a simplified scenario, departing passengers check in and board the plane, during the flight they might consume a meal, and after the plane has arrived at the destination airport, they deplane and claim their luggage. The

complete dynamic behavior of the objects of classes Passenger, Airport and Plane was modeled in UMC in the form of statechart diagrams and subsequently a number of logical properties were verified [2].

The experimentation with UMC has continued also in the context of the SENSORIA project, where it was used to model and verify an asynchronous version of the SOAP communication protocol. In the same project, UMC has been used for the modeling of an automotive scenario, for the support of the SRML modeling language, and for the conflict detection of policies in a scenario from SENSORIA's Finance case study [15, 1, 20, 8].

More recently, UMC was successfully applied, in the context of the regional project PAR-FAS-2007-2013 TRACE-IT (Train Control Enhancement via Information Technology), to the development of a model-checking-based design methodology for deadlock-free train scheduling [52, 51].

5 CMC: Parametrized Logic Formulas for Expressing Data Correlations Among Actions

A third application of our on-the-fly model-checking approach has been to the process algebra COWS [44, 56], developed in the context of the SENSORIA project. This project developed a novel comprehensive approach to the engineering of software systems for SOC. Foundational theories, techniques and methods were fully integrated in a pragmatic software engineering approach that focused on global services that are context-adaptive, personalizable, and which may require hard and soft constraints on resources and performance. Moreover, the fact that services have to be deployed on different, possibly interoperating global computers to provide novel and reusable service-oriented overlay computers was taken into account.

The Calculus for Orchestration of Web Services (COWS) is a modeling notation for all relevant phases of the life cycle of service-oriented applications, among which service publication, discovery, and orchestration, as well as Service-Level Agreement (SLA) negotiation. Besides service interactions and compositions, important aspects like fault and compensation handling can be modeled in COWS. Extensions moreover allow timed activities, constraints and stochastic reasoning. Application to the SENSORIA case studies [8] has demonstrated the feasibility of modeling service-oriented applications with the specific mechanisms and primitives of COWS [35].

Experimentation in this direction led to the development of the CMC model checker for COWS terms and the definition of the SocL logic [24, 35]. It is too complex to explain in detail all the features and characteristics of the COWS specification language. Here we only mention that COWS is a process-algebraic language that allows recursive processes which can also be parallel process (unlike FMC, which does not not allow parallelism inside recursion). Process synchronization and communication occurs through input/output actions which have the form $p.o! < args >$ and $p.o? < params >$ where p denotes a communication partner and o an operation request. Recursion is achieved through a 'bang'

operator ($*P$) meaning $P|P|P|\cdots$. The language supports also the definition of protected contexts ($\{P\}$), delimited contexts ($[k]\,P$) and kill operations ($kill(k)$).

This kind of systems require a logic that allows to express the *correlation* between dynamically generated values appearing inside actions at different times. The reason for this is that such correlation values then allow, *e.g.*, to relate the responses of a service to their specific request, or to handle the concept of a session involving a long sequence of interactions among the interacting partners. A typical example property that one would like to express in this context is that whenever a process performs a *request* operation to a partner p, providing some identification data id, in all cases the partner will reply with a *response* operation with the same identification data. In CMC that property can be expressed by the parametric formula:

$$AG\ [request(p, \$id)]\ AF\ response(p, \%id)\ true$$

CMC has been successfully used to model and analyze service-oriented scenarios from the SENSORIA project's Automotive and Finance case studies and to its Bowling Robot case study [19, 15, 20, 14, 21, 8, 35].

6 VMC: Behavioral Variability Analysis for Product Families

The final and most recent extension of the modeling and verification framework that we will describe in this paper is a tool, called the Variability Model Checker (VMC [25, 16, 23]), which was specifically developed for the specification and verification of so-called *product families* or *product lines*.

Software Product Line Engineering (SPLE) [30, 55] is by now an established field of software-intensive system development which propagates the systematic reuse of assets or features in an attempt to lower production costs and time-to-market and to increase overall efficiency. SPLE thus aims to develop, in a cost effective way, a variety of software-intensive products that share an overall reference model, *i.e.* that together form a product family. Usually, *commonality* and *variability* are defined in terms of so-called *features*, and managing variability is about identifying variation points in a common family design and deciding which combinations of features are to be considered valid products. There is by now a large body of literature on the computer-aided analysis of feature models to extract valid products and to detect anomalies, *i.e.* undesirable properties such as superfluous or—worse—contradictory variability information (*e.g.* so-called false optional or dead features) [26].

Until a few years ago, these analyses however did not take any behavior into account, even though software products are often large and complex, and many are used in safety-critical applications in the avionics, railways, or automotive industries. The importance of specifying and verifying also *behavioral variability* was first recognized in the context of UML [43, 58]. Shortly after, in [37], Modal Transition Systems (MTSs) were recognized as a promising formal method for describing in a compact way the possible operational behavior of the products in

a product family. An MTS [3] is a Labeled Transition System (LTS) distinguishing between 'admissible' *may* and 'necessary' *must* transitions. By definition, every must transition is also a may transition.

In recent years, many variants and extensions of MTSs have been studied in order to elaborate a suitable formal modeling structure to describe (behavioral) variability [34, 45, 46, 4–6]. This has resulted in a growing interest in modeling behavioral variability in general, which has led to the application of a number of formal methods different from MTSs but still with a transition system semantics [42, 54, 41, 47, 53, 22, 29, 10–12, 48, 28]. As a consequence, behavioral analysis techniques like model checking have become available for the verification of (temporal) logic properties of product families.

VMC accepts the specification of an MTS in process-algebraic terms, together with an optional set of additional variability constraints. VMC then allows to perform the following two kinds of behavioral variability analyses on a given family of products:

1. A logic property expressed in a *variability-aware* version of ACTL (v-ACTL) can directly be verified against the MTS modeling the product family behavior, relying on the fact that under certain syntactic conditions the validity of the property over the MTS guarantees the validity of the same property for all products of the family.
2. The actual set of valid product behavior can explicitly be *generated* and the resulting LTSs can be verified against the same logic property (expressed in ACTL). This is surely less efficient than direct MTS verification but allows to precisely identify the set of features whose interactions may cause the original property to fail over the whole family.

The process algebra used by VMC to specify the MTS modeling of the behavior of a product family is derived from the one of FMC by removing CCS-like synchronizations and adding to the actions the notion of variability. In fact, in VMC, communication/synchronization actions can accept an additional parameter (*may*) which expresses the property that the action is not necessarily present in all derivable products of the family. The synchronization semantics is also updated by taking this parameter into consideration, in the sense that the result of the synchronization of an optional action with a mandatory action results in an optional action [3].

In more detail, the structure of the process algebra accepted by VMC is described by the following abstract syntax:

$$\begin{aligned} &\mathit{System} \ ::= \ [\,\mathit{Net}\,] \\ &\mathit{Net} \ ::= \ T(\mathit{expr}) \ \mid \ \mathit{Net} \ /\, \mathit{Labels} \,/ \ \mathit{Net} \end{aligned}$$

where $[\,\mathit{Net}\,]$ denotes again a closed system, $T(\mathit{expr})$ is a process instantiation from the set of process declarations of the form $T(\mathit{variable}) \stackrel{\text{def}}{=} \mathit{Term}$, and *Labels* is a list of action names.

The structure of *Term* definitions is described by the following abstract syntax:

$$
\begin{array}{l}
\mathit{Term} \ ::= \ \mathit{nil} \ \mid \ T(\mathit{expr}) \ \mid \ \mathit{Action}.\mathit{Term} \ \mid \ \mathit{Term} + \mathit{Term} \ \mid \ [\,\mathit{expr} \bowtie \mathit{expr}\,]\,\mathit{Term} \\
\mathit{Action} \ ::= \ a(\mathit{arg}) \ \mid \ a(\mathit{may}, \mathit{arg}) \\
\mathit{arg} \ ::= \ \mathit{expr} \ \mid \ ?\mathit{variable} \\
\mathit{expr} \ ::= \ \mathit{variable} \ \mid \ \mathit{integer} \ \mid \ \mathit{expr} \pm \mathit{expr}
\end{array}
$$

where $\bowtie$ is a comparison operator and $\pm$ is an arithmetic operation.

In VMC, the abstract model associated to this variability-oriented process algebra is an LTS in which edges are labeled with sets of labels, and where the additional *may* label is added to the optional edges to specify their possible absence in some of the family's products.

The logic v-ACTL is built over a subset of ACTL, but enriched with the *deontic* operators $AF\#$, $EF\#$, $\langle\rangle\#$, and $[]\#$ (*cf.* [18, 23] for details). These operators are actually implemented in VMC by a translation into plain ACTL. For example, the formula $\langle a\rangle\#\ \mathit{true}$, which means that there exists a mandatory evolution from the current state which satisfies action a, can be encoded in plain ACTL as $\langle a\ \mathit{and\ not\ may}\rangle\ \mathit{true}$. Similarly, $EF\#\ \phi$ can be encoded in plain ACTL as $E\,[\,\mathit{true}\ \{\mathit{not\ may}\}\ U\ \phi\,]$ (where ϕ is a subformula).

We are currently experimenting with VMC in the context of the European FP7-ICT-600708 project QUANTICOL [9] (*cf.* `http://www.quanticol.eu`). So far the case studies taken into consideration (a bike-sharing system and a coffee machine [13, 18]) are relatively small and more effort is needed to evaluate the approach on problems of a more realistic size.

7 The Overall Structure of the Model Checkers

In the previous sections, we have seen four different specification languages for the four model checkers that are part of KandISTI. While their computational models are rather different, ranging from statecharts to various kind of process algebras, the evolution of the framework over time has led to the development of a *unique* common temporal logic and verification engine, which encompasses and integrates the various specific logics initially associated to the specific tools: ACTL for FMC, UCTL for UMC, SocL for CMC and v-ACTL for VMC.

This had become feasible by splitting the statespace generation problem (which depends on the underlying computational model), from the L^2TS analysis problem, and by the introduction of an explicit abstraction mechanism which allows to specify which details of the model should be observable as labels on the states and transitions of the L^2TS.

Another essential characteristic of our family of tools, which has been preserved since its origins, is the so-called *on-the-fly* structure of the model-checking algorithm: the L^2TS corresponding to the model is generated *on-demand*, following the incremental needs of the logical verification engine. Given a state of an L^2TS, the validity of a logic formula on that state is evaluated by analyzing

the transitions allowed in that state, and by analyzing the validity of the necessary subformulae possibly in some of the necessary next reachable states, all this recursively.

Indeed, each tool consists of two separate, but interacting, components: a tool-specific L^2TS generator engine and a common logical verification engine. The L^2TS generator engine is again structured in two logical components: a ground evolutions generator, strictly based on the operational semantics of the language, and an abstraction mechanism which allows to associate abstract observable events to system evolutions and abstract atomic propositions to the system states. The verification engine is the component which actually tries to evaluate a logic formula following the on-the-fly approach, and is described in more detail in [14, 35].

The L^2TS generator engine maintains an archive of already generated system states in order to avoid unnecesary duplications in the computation of the possibile evolutions of states. The logical verification engine maintains an archive of logical computation fragments; this is not only useful to avoid unnecessary duplications in the evaluation of subformulae, but also necessary to deal with the recursion in the evaluation of a formula arising from the presence of loops in the models. The overall structure of the framework is shown in Fig. 3.

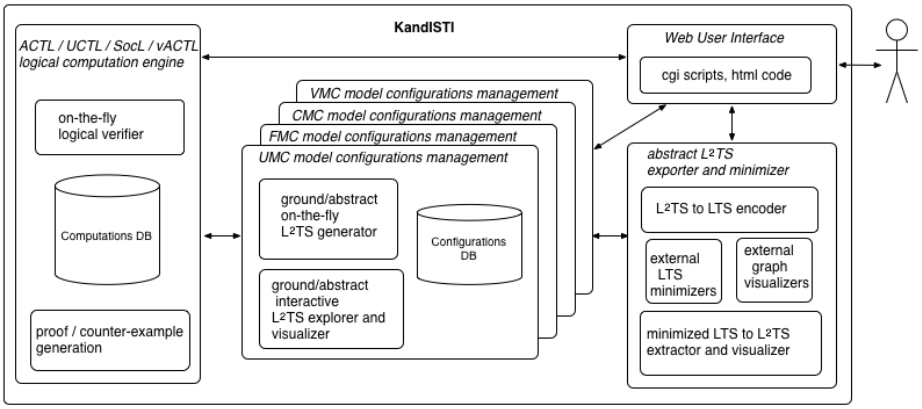


Fig. 3. The architecture of the KandISTI framework

All the model checkers of our family are constituted by a command-line version of the tool written in Ada, which can be easily compiled for the Windows, Linux, Solaris and Mac OS X platforms. These core executables are wrapped with CGI scripts handled by a web server, facilitating an html-oriented GUI and integration with graph drawing tools. It is beyond the scope of this paper to give detailed descriptions of the model-checking algorithms and architecture that underly our family of model checkers. Instead, we refer the interested reader to [24, 40, 14, 35] for more details.

8 Discussion and Conclusions

In this paper, we have provided an overview of the KandISTI family of model-checking tools, currently consisting of FMC, UMC, CMC and VMC. We have briefly presented their different kind of underlying computational models and their different contributions to the development of a general purpose action- and state-based branching-time temporal modal logic (with special purpose dialects for each of the KandISTI variants).

The differences between the described input models stem from the specific field of application for which they were developed. In the end, however, each of them is interpreted over an L^2TS, which permits to use the same logical verification engine for all, even though the specific logic associated to each of the input models again has certain features that are specifically tailored towards the application field for which they were developed.

FMC's input model of automata networks was defined as an attempt to integrate the communication and synchronization mechanisms from CCS, CSP and LOTOS in a single process-algebra, thus allowing both multi-way synchronization and value-passing.

UMC's input model of UML-like state machines was inspired by the UML paradigm of dynamic systems seen as sets of evolving and communicating objects, where objects are class instances. In UML, the set of objects and classes constituting a system are described by a structure diagram, while the objects' dynamic behaviour is described by associating a statechart diagram to their classes. As a result, each system object behaves like a state machine, with a set of local attributes, an event pool collecting the events to be processed, and the currently active states of the state diagram.

CMC's input model COWS was influenced by WS-BPEL principles for Web service orchestration, thus supporting the correlation of different actions and the management of long-running transactions. The pure process-algebraic specification in COWS terms needs to be accompanied with a set of abstractions that define the action semantics and state predicates.

VMC's input model of process-algebraic interpretations of MTSs, possibly enriched with variability constraints known from SPLE, was developed to study the feasibility of using MTSs to describe (and consequently analyze) in a compact way the possible operational behaviour of products from a product family.

The KandISTI model checkers are continuously being improved. This ranges from more efficient generation of the computational models to a more user-friendly web interface. An overall goal for the future is to experiment with industrial case studies of increasing size. In order to fulfill this aim, richer input models are needed, in particular allowing more advanced data types (*e.g.* tuples, sets, lists, *etc.*, currently only supported by UMC), thus requiring more complex computational models. The addition of some kind of global data space shared among the concurrent objects/agents might moreover be a useful extension to more easily support also the underlying models of other verification frameworks, like Spin or SMV.

Future work that is specifically related to VMC concerns studying the specific fragment of v-ACTL that is guaranteed to be preserved by product refinement. This would allow built-in user notification in all cases in which a model-checking result is guaranteed to be preserved from family to product level.

As already hinted at in Section 4, the main lesson learned by the use of our framework is the usefulness of an easy-to-use formal framework for the early analysis of initial system designs, i.e. the usefulness of formal methods in the earliest stages of system design, when the first architectural/algorithmic ideas are being prototyped and debugged. This is a very different application of formal methods with respect to their classical use in the final validation/verification steps, when the system is already supposed to be (hopefully) free of errors. In the former case, it is important to be able to rely on formal frameworks which simplify and make the task of modeling and debugging a system (which with a high probability is expected to contain errors) easy, while in the latter case the emphasis can be put on the power to deal with extremely large state spaces in a very efficient way.

We believe that KandISTI can successfully match the needs of agile formal designers (which constitute its natural set of target users) while still not disregarding the issues introduced by the problems of the possible state space explosions.

Acknowledgements

Major progress on KandISTI was made during almost a decade of EU projects under the inspiring coordination of Martin Wirsing. We would like to take this opportunity to thank him for the work we did together.

The three paintings by Wassily Kandinsky that are part of the collection of the Städtische Galerie im Lenbachhaus in Munich, Germany, and whose images are used here for non-commercial strictly illustrative purposes, have entered the public domain in the EU on January 1st, 2015 (70 years *post mortem auctoris*, imposed by Article 1 of EU Directive 93/98/EEC as repealed and replaced by EU Directive 2006/116/EC).

References

1. J. Abreu, F. Mazzanti, J. L. Fiadeiro, and S. Gnesi. A Model-Checking Approach for Service Component Architectures. In *FMOODS'09*, volume 5522 of *LNCS*, pages 219–224. Springer, 2009.
2. L. F. Andrade et al. AGILE: Software Architectures for Mobility. In *WADT*, volume 2755 of *LNCS*, pages 1–33. Springer, 2003.
3. A. Antonik, M. Huth, K. G. Larsen, U. Nyman, and A. Wąsowski. 20 Years of Modal and Mixed Specifications. *Bulletin of the EATCS*, 95:94–129, 2008.
4. P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. A Model-Checking Tool for Families of Services. In *FMOODS*, volume 6722 of *LNCS*, pages 44–58. Springer, 2011.

5. P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. Formal Description of Variability in Product Families. In *SPLC*, pages 130–139. IEEE, 2011.
6. P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. A Compositional Framework to Derive Product Line Behavioural Descriptions. In [49], pages 146–161.
7. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
8. M. H. ter Beek. Sensoria Results Applied to the Case Studies. In [57], pages 655–677.
9. M. H. ter Beek, L. Bortolussi, V. Ciancia, S. Gnesi, J. Hillston, D. Latella, and M. Massink. A Quantitative Approach to the Design and Analysis of Collective Adaptive Systems for Smart Cities. *ERCIM News: Smart Cities*, 98:32, July 2014.
10. M. H. ter Beek and E. P. de Vink. Software Product Line Analysis with mCRL2. In [38], pages 78–85.
11. M. H. ter Beek and E. P. de Vink. Towards Modular Verification of Software Product Lines with mCRL2. In [50], pages 368–385.
12. M. H. ter Beek and E. P. de Vink. Using mCRL2 for the analysis of software product lines. In *FormaliSE*, pages 31–37. IEEE, 2014.
13. M. H. ter Beek, A. Fantechi, and S. Gnesi. Challenges in Modelling and Analyzing Quantitative Aspects of Bike-Sharing Systems. In [50], pages 351–367.
14. M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 76(2):119–135, 2011.
15. M. H. ter Beek, S. Gnesi, N. Koch, and F. Mazzanti. Formal verification of an automotive scenario in service-oriented computing. In *ICSE*, pages 613–622. ACM, 2008.
16. M. H. ter Beek, S. Gnesi, and F. Mazzanti. VMC: A Tool for the Analysis of Variability in Software Product Lines. *ERCIM News: Mobile Computing*, 93:50–51, January 2013.
17. M. H. ter Beek, S. Gnesi, and F. Mazzanti. KandISTI: A Family of Model Checkers for the Analysis of Software Designs. *ERCIM News: Software Quality*, 99:31–32, October 2014.
18. M. H. ter Beek, S. Gnesi, and F. Mazzanti. Model Checking Value-Passing Modal Specifications. In *PSI*, LNCS. Springer, 2014. To appear.
19. M. H. ter Beek, S. Gnesi, F. Mazzanti, and C. Moiso. Formal Modelling and Verification of an Asynchronous Extension of SOAP. In *ECOWS*, pages 287–296. IEEE, 2006.
20. M. H. ter Beek, S. Gnesi, C. Montangero, and L. Semini. Detecting policy conflicts by model checking UML state machines. In *ICFI*, pages 59–74. IOS Press, 2009.
21. M. H. ter Beek, A. Lapadula, M. Loreti, and C. Palasciano. Analysing Robot Movement Using the Sensoria Methods. In [57], pages 678–697.
22. M. H. ter Beek, A. Lluch-Lafuente, and M. Petrocchi. Combining declarative and procedural views in the specification and analysis of product families. In *SPLC*, volume 2, pages 10–17. ACM, 2013.
23. M. H. ter Beek and F. Mazzanti. VMC: Recent Advances and Challenges Ahead. In [38], pages 70–77.
24. M. H. ter Beek, F. Mazzanti, and S. Gnesi. CMC–UMC: a framework for the verification of abstract service-oriented properties. In *SAC*, pages 2111–2117. ACM, 2009.
25. M. H. ter Beek, F. Mazzanti, and A. Sulova. VMC: A Tool for Product Variability Analysis. In *FM*, volume 7436 of *LNCS*, pages 450–454. Springer, 2012.
26. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Information Systems*, 35(6), 2010.

27. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
28. A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming*, 80(B):416–439, 2014.
29. A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, 2013.
30. P. C. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
31. R. De Nicola, editor. *ESOP*, volume 4421 of *LNCS*. Springer, 2007.
32. R. De Nicola and F. W. Vaandrager. Action versus State based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.
33. R. De Nicola and F. W. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.
34. A. Fantechi and S. Gnesi. A behavioural model for product families. In *ESEC/FSE*, pages 521–524. ACM, 2007.
35. A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A logical verification methodology for service-oriented computing. *ACM Transactions on Software Engineering and Methodology*, 21(3):16, 2012.
36. C. Fidge. A Comparative Introduction to CSP, CCS and LOTOS. Technical Report 93-24, Software Verification Research Centre, University of Queensland, January 1994.
37. D. Fischbein, S. Uchitel, and V. A. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA*, pages 39–48. ACM, 2006.
38. S. Gnesi, A. Fantechi, M. H. ter Beek, G. Botterweck, and M. Becker, editors. *Proceedings of the 18th International Software Product Line Conference (SPLC'14)*, volume 2. ACM, 2014.
39. S. Gnesi and F. Mazzanti. On the Fly Verification of Networks of Automata. In *PDPTA*, pages 1040–1046. CSREA Press, 1999.
40. S. Gnesi and F. Mazzanti. An Abstract, on the Fly Framework for the Verification of Service-Oriented Systems. In [57], pages 390–407.
41. S. Gnesi and M. Petrocchi. Towards an executable algebra for product lines. In *SPLC*, volume 2, pages 66–73. ACM, 2012.
42. A. Gruler, M. Leucker, and K. Scheidemann. Modeling and Model Checking Software Product Lines. In *FMOODS*, volume 5051 of *LNCS*, pages 113–131. Springer, 2008.
43. Ø. Haugen and K. Stølen. STAIRS: Steps to Analyze Interactions with Refinement Semantics. In *UML*, volume 2863 of *LNCS*, pages 388–402. Springer, 2003.
44. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In [31], pages 33–47.
45. K. G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O Automata for Interface and Product Line Theories. In [31], pages 64–79.
46. K. Lauenroth, K. Pohl, and S. Töhning. Model Checking of Domain Artifacts in Product Line Engineering. In *ASE*, pages 269–280. IEEE, 2009.
47. M. Leucker and D. Thoma. A Formal Approach to Software Product Families. In [49], pages 131–145.
48. M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. DeltaCCS: A Core Calculus for Behavioral Change. In [50], pages 320–335.

49. T. Margaria and B. Steffen, editors. *ISoLA*, volume 7609 of *LNCS*. Springer, 2012.
50. T. Margaria and B. Steffen, editors. *ISoLA*, volume 8802 of *LNCS*. Springer, 2014.
51. F. Mazzanti, G. O. Spagnolo, S. Della Longa, and A. Ferrari. Deadlock Avoidance in Train Scheduling: a Model Checking Approach. In *FMICS*, volume 8718 of *LNCS*, pages 109–123. Springer, 2014.
52. F. Mazzanti, G. O. Spagnolo, and A. Ferrari. Designing a Deadlock-Free Train Scheduler: A Model Checking Approach. In *NFM*, volume 8430 of *LNCS*, pages 264–269. Springer, 2014.
53. J.-V. Millo, S. Ramesh, S. N. Krishna, and G. K. Narwane. Compositional Verification of Software Product Lines. In *IFM*, volume 7940 of *LNCS*, pages 109–123. Springer, 2013.
54. R. Muschevici, J. Proença, and D. Clarke. Modular Modelling of Software Product Lines with Feature Nets. In *SEFM*, volume 7041 of *LNCS*, pages 318–333. Springer, 2011.
55. K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
56. R. Pugliese and F. Tiezzi. A Calculus for Orchestration of Web Services. *Journal of Applied Logic*, 10(1):2–31, 2012.
57. M. Wirsing and M. M. Hölzl, editors. *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, volume 6582 of *LNCS*. Springer, 2011.
58. T. Ziadi and J.-M. Jézéquel. Software Product Line Engineering with the UML: Deriving Products. In *Software Product Lines: Research Issues in Engineering and Management*, pages 557–588. Springer, 2006.