A Language Agnostic Approach to Modeling Requirements: Specification and Verification
Abdelghani Alidra, Antoine Beugnard, Hubert Godfroy, Pierre Kimmel, Gurvan Le Guernic

To cite this version:

HAL Id: hal-02924645
https://inria.hal.science/hal-02924645
Submitted on 28 Aug 2020
ABSTRACT

Modeling is a complex and error-prone activity which can result in ambiguous models containing omissions and inconsistencies. Many works have addressed the problem of checking models' consistency. However, most of these works express consistency requirements for a specific modeling language. On the contrary, we argue that in some contexts those requirements should be expressed independently from the modeling language of the models to be checked. We identify a set of modeling requirements in the context of embedded systems design that are expressed independently from any modeling language concrete syntax. We propose a dedicated semantic domain to support them and give a formal characterization of those requirements that is modeling language agnostic.

CCS CONCEPTS
• Software and its engineering → Model-driven software engineering; System modeling languages; • Computing methodologies → Model verification and validation.

KEYWORDS
Modeling, domain specific, modeling requirements, validation, formal definition, system decomposition

ABDELGHANI ALIDRA
Antoine Beugnard
alidrandco@yahoo.fr
antoine.beugnard@imt-atlantique.fr
IMT Atlantique Bretagne-Pays de la Loire
Brest, France

HUBERT GODFROY
Pierre Kimmel
hubert.godfroy@capgemini.com
pierre.kimmel@capgemini.com
Capgemini France
Cesson-Sévigné, France

GURVAN LE GUERNIC
gurvun.le-guernic@def.gouv.fr
gurvun.le_guernic@inria.fr
DGA Maîtrise de l’Information
Inria Rennes – Bretagne Atlantique
Rennes, France

A Language Agnostic Approach to Modeling Requirements:
Specification and Verification

ABDELGHANI ALIDRA
Antoine Beugnard
alidrandco@yahoo.fr
antoine.beugnard@imt-atlantique.fr
IMT Atlantique Bretagne-Pays de la Loire
Brest, France

HUBERT GODFROY
Pierre Kimmel
hubert.godfroy@capgemini.com
pierre.kimmel@capgemini.com
Capgemini France
Cesson-Sévigné, France

GURVAN LE GUERNIC
gurvun.le-guernic@def.gouv.fr
gurvun.le_guernic@inria.fr
DGA Maîtrise de l’Information
Inria Rennes – Bretagne Atlantique
Rennes, France

1 INTRODUCTION

Models can play a pivotal role in the development process of embedded systems in a contractual relationship. Models can be leveraged during many development phases, in particular during evaluation procedures: in the design phase; or in the validation phase, or even in a certification phase. However, experience shows that modeling is hard and non trivial. Models represent systems at different abstraction levels and from different points of view. They are intended for various stakeholders and for many different tasks. It is therefore important in a contractual relationship that modeling (completeness) requirements clearly elicit what information needs to be part of the models. Then, during the description of large systems, many modeling elements are defined. Among those elements, many relations, dependencies and constraints may exist. Consequently, errors, omissions and inconsistencies can easily arise. In order to be exploitable, models need to avoid such defects. It is then critical to track and correct those errors. Which in turn requires to elicit those constraints and correctness properties, potentially as modeling (soundness) requirements.

Many existing works have addressed this challenge in the literature, mostly the soundness verification part and less the completeness elicitation part. Focusing on UML models, early works attempted to check consistency of class diagrams [5, 41, 58]. These works were extended to handle all or part of OCL constraints. Some works map UML/OCL models to formal specifications expressed in CSP, ObjectZ or B languages [11, 42, 51]. Other approaches express the consistency of UML/OCL models as a satisfiability problem written in relational logic [3, 25, 35, 56], in first order logic [14, 15] or in the HOL/Isabelle language [35]. Some works focused on providing assistance to find conflicts to help modelers debugging inconsistent UML/OCL models [61–63]. Other approaches were interested in checking consistency of models expressed in other modeling languages. For instance, Weckesser et al. address the problem of checking the consistency of Clafier models [59, 60] and Bertram et al. check the consistency of Component and Connectors view models [6, 40]. As a last example, the work introduced by Schrefl and Stumpter [55] presents the particularity of checking behavioral models of the system expressed as behavior diagrams.

A notable characteristic common to the previously cited approaches is that they are closely bound to the language or formalism that is used to model the system under study. However, different companies developing embedded systems may use different modeling languages, such as SysML [44], SDL [32], AADL [20] or even OPM [29] or ARCADIA/Capella [53] to name a few. Hence, modeling requirements expressed by contracting or certification authorities should ideally apply equally to all those modeling languages. Contrary to the previously cited approaches, we aim at exploring a modeling language agnostic approach to express (completeness and soundness) modeling requirements, i.e. our approach defines modeling requirements independently from any concrete
modeling language. This allows contracting or certification authorities to impose requirements on delivered models while minimizing the constraints imposed on contractors or vendors regarding the use of their preferred modeling language or development tools and processes.

To cope with this constraint, we follow the agnostic principle advocated for by Le Guernic [37] and introduce a new abstract semantic domain\(^1\) to formally express modeling requirements in an agnostic way. Some parts of this semantic domain definition bears resemblance to IO automata [33, 39]. We also define a set of language agnostic constraints to verify these modeling requirements. In this paper, we focus on requirements related to the functional decomposition of the system. The focus on these particular requirements is motivated by two complementary aspects. Firstly, these requirements reflect the need of procurement agencies to cover some specific verification aspects of the systems, essentially those related to the validation of security equipment. Thus, based on our partners’ expertise, these requirements naturally emerged as a prominent and significant subset for this kind of activity. Secondly, as can be noticed from the quick overview presented above, lots of works have been focusing on structural and static models while consistency checking of dynamic and behavioral models has been less considered.

In particular, our current interest is on requirements regarding the behavioral equivalence between: the system; its decomposition; system views; and the environmental solicitations. Indeed, in a contractual or certification relationship, and because of the complexity of the considered systems, it is beneficial if contractors provide (semi) formal models that describe: the global behavior of the system; a first level of behavioral decomposition; and a set of functional views that focus on particularly important or sensitive features of the system. Obviously, the behavioral decomposition and the functional views must be consistent with the global system specification in the sense that the behavior of the system is preserved. Besides, it is important that the models delivered explicit the behavior of the system with regard to environmental solicitations that are relevant for the analysis that has to be made. The verification of those completeness and soundness modeling requirements is a prerequisite to analyze the models further. These verifications are for model analysis what smoke tests [19] are for system testing.

The contributions of this paper are as follows. First, Sect. 2 introduces a specific semantic domain to express language agnostic modeling requirements related to structural and behavioral decomposition, which is exploited in Sect. 3 to formally define, in a modeling language agnostic way, a set of completeness and soundness structural and behavioral decomposition modeling requirements. Related work is discussed in Sect. 4 before concluding.

2 A NEW SEMANTIC DOMAIN FOR ASSESSING MODELS CONSISTENCY REQUIREMENTS

In this section, we introduce a semantic domain for a new family of models we call Behavioral Decomposition Models (BDMs). We present an overview of our design choices and of the general structures of BDMs focusing on some particular sub-models. Section 3 focuses on some formal verification we proposed on BDMs.

2.1 Structure

A BDM of a system sys in an environment env is a formally defined model consisting in several sub-models:

- (1) An Interface Model (IM) listing the interfaces of sys, the messages exchanged between sys and env and the mapping of those messages on the interfaces through events.
- (2) An Environment Model (EM) listing the solicitations by which env can challenge sys which are taken into account in the current model.
- (3) A System Functions Model (SFM) listing the functions that sys can execute (at black box level).
- (4) A System Behaviour Model (SBM) describing how sys reacts to the solicitations from env.
- (5) A set of Functional View Models (FVMs) describing the behavior of sys focusing on some messages or functions.
- (6) A System Organs Model (SOM) listing all sys organs (ie its first level logical and/or structural components) as well as the structure of these organs: which interfaces they use, which messages they exchange with each other and which functions they can execute.
- (7) A set of Functional Organic Models (FOMs) describing the decomposition of black box level functions into calls of organic functions as well as the messages these functions use to communicate.
- (8) A set of Organic Behaviour Models (OBMs) describing the behavior of each organ, at a black box level.

We can see that: sub-models 1 to 4 describe the black box behavior of sys and its interactions with env; with sub-model 5 providing further abstractions of the black box behavior; whereas sub-models 6 to 8 describe the organic decomposition and behavior of sys (first level of decomposition).

2.2 Events and solicitations

We use the notion of event to describe interactions between sys and env, as well as between organs. An event is a triple \((\text{int}, \text{msg}, \text{fl})\) where \(\text{int}\) is an interface, \(\text{msg}\) is a message and \(\text{fl}\) is a flow direction (either in or out). This allows the same message (resp. interface) to be used in several interfaces (resp. with several messages). The flow direction allows us to specify interfaces that are both input and output interfaces. We note \(\text{Ev}_{\text{sys}}\) the set of events between sys and env.

Given some sets of events noted \(E\), we define the set of solicitations \(\text{Sol}\) by: \(\text{Sol} \subseteq \{ \{\text{or}(E)\text{not}(E)\}^* \mid E \subseteq \text{Ev}_{\text{sys}} \}\).

A solicitation is a sequence of sets of events, each set being in the scope of an operator or an operator not. In each set, the events are all tagged with either in or out. Solicitations are a completeness requirement stating for which possible solicitations BDM must explicit the behavior of sys. The intuitive semantics of such sets is as follows:

- A set of in-events in the scope of or means that env solicits sys with one of these events. BDM must explicit the behavior

---

\(^1\)Due to lack of space, the mapping of General Purpose Modeling Languages, such as SysML, to this semantic domain is not described in this paper.
of sys for at least one of these events at this point of the solicitation.
- A set of out-events in the scope of or means that env will wait for at least one of these events from sys in order to continue the solicitation. If sys produces such output, BD must explicit the behavior of sys for the rest of the solicitation.
- A set of out-events in the scope of not means that env will not continue the solicitation after receiving this answer. As long as sys produces no such output and does not react to another event, BD must explicit the behavior of sys for the rest of the solicitation.

For instance, the following solicitation: $S = \text{or}(e_1^i, e_2^i) \cdot \text{not}(e_3^o) \cdot \text{or}(e_4^o) \cdot \text{or}(e_5^o)$ where the $e_1^i$ are input events and $e_2^i$ are output events, means that BD must explicit the behavior of sys from every initially reachable state, for at least a case where: env sends either $e_1^i$ or $e_2^i$, then, from every state reachable without additional environment solicitation where sys has replied $e_5^o$ without replying $e_1^i$ before, for at least a case where env sends $e_2^i$.

In some way, the set of solicitations contained in the BDM model describes all the scenarios for which we want our model to explicit the behavior of the system.

### 2.3 Behavior

The behavior of sys, described in BDM is a set of executions, that is to say transitions from system state to system state. Each transition is labeled by:

- at most one input event $e^i$ representing the input event that can trigger the transition,
- a set $F$ of functions representing the functions that sys executes when going through this transition,
- a set $E$ of output events representing the outputs produced by sys when going through this transition.

In the general case, we will note $e^i/F/E$ the label of such a transition. In the case where no event triggers the transition, we will note $\bot$ instead of $e$.

Note that this structure does not indicate the order of execution and emission between the functions of $F$ and the messages of $E$: we just state that such functions and messages are produced, possibly those messages are the result of these functions, but it is not guaranteed by the model. Similarly, we can suppose that the triggering event $e^i$ caused functions and messages to be produced, but it is not guaranteed. This nondeterminism is necessary in order to allow some implementation freedom while designing the system, and perform retro-engineering.

In paragraph 3.1, we will explain how we propose to check that the behavior of sys is coherent with Sol (meaning it always specifies a behavior for each of those solicitations).

This semantic domain for behaviors bear resemblance to the semantic domain of automata, notably of Input/Output (IO) automata [17, 33, 36, 39] and particularly partial order IO automata [7]. However, for those semantic domains, the set of executions is a secondary entity derived from an automaton, whereas in our case the set of executions is a primary entity in the semantic domain, for which there may not be a corresponding automaton. In particular, the set of executions of an automaton is closed by “concatenation”, which is not the case for the set of executions of our semantic domain. This is needed for cases where the reason for an execution to be possible or not is not explicit at the level of abstraction used for sys’s behavior specification, or if sys’s behavior is syntactically specified by a set of sequence diagrams [43, §17.8 p.595] or message sequence charts [30, 31]. Moreover, the general meaning of an automaton’s executions set is that, if an execution does not belong to this set then this execution is not a possible execution of the system represented by the automaton. In our semantic domain, if an execution $x$ does not belong to sys’s executions set, then it only means that the fact that sys can or can not show behavior $x$ is unspecified.

If one wants to specify that “$x$” is not a possible execution of sys, then the error “handling” behavior of sys on $x$ must be explicitly specified and “$x$” needs to be explicitly added to sys’s executions set with final states being error states, while $x$ must be “covered” by a solicitation in Sol (this completeness aspect is detailed in Sect 3.1).

However, in this paper and for conciseness purposes, the syntax used to define this set will often be an automaton.

### 2.4 Views

Due to the complexity of some systems, it is necessary to have tools to focus on some particular aspects of it. We make use of the notion of views. Each FVM describes the behavior of sys in the same way that BDM did, except that it removes or merges some states and/or transitions according to certain input messages of interest and functions of interest. The effects and executability of these items of interest have to be emphasized by those FVMs.

For now, our work does not provide methods to design such views but it must check that:

- the behavior of sys according to the FVM does not contradict the behavior described by BDM;
- FVM contains the exact necessary amount of information to study the items of interest.

To check this, we use 4 properties characterizing relationships between a trace of execution $t$ and a function $f$:

- **Permission**: $t$ allows $f$ if $t$ ends with $f$ and is minimal (there is no subtrace of it leading to $f$);
- **Requirement**: $t$ is required for $f$ if it is a subtrace of every trace allowing $f$ and it is maximal (there is no trace having $t$ as a subtrace that verifies this);
- **Inhibition**: $t$ inhibits $f$ if there is a trace $t'$ leading to $f$ and adding $t$ into $t'$ does not lead to $f$ and it is minimal (temporary disabling);
- **Blocking**: $t$ blocks $f$ if there is a trace $t'$ leading to $f$ and adding $t$ into $t'$ cannot be extended into a trace leading to $f$ and it is minimal (permanent disabling).

---

2 If the two executions $x_{1a}, x_{1b}$ and $x_{2a}, x_{2b}$ belong to the executions of an automaton $A$, and $x_{1a}$ and $x_{1b}$ finish in the same state where $x_{2a}$ and $x_{2b}$ start, then $x_{1a}, x_{2b}$ and $x_{2a}, x_{2b}$ are also executions of $A$.

3 In the context of the dialog between project management and designers, managers would describe the item of interest they want to examine and designers would use their knowledge of the (future) product to produce the view models.

4 Subtrace is an order relation akin to scattered subwords.

5 As a scattered subword.

6 $t$ extends $t'$ if $t'$ is a subtrace of $t$. 
We then use those properties both with traces containing events of interest and functions of interests, and we ensure that the view preserves those properties with regards to the general behavior of the system. We also reach minimality by ensuring that only elements relevant to those properties appear in the view.

We will not give any further details on this method in the scope of this article. The rigorous definition of system views is complex enough to be the topic of another publication on its own.

2.5 Organic decomposition
The decomposition of sys is described by several models. We call organ each component of sys studied by our model.7

The interfaces, functions and behavior of each organ is described in the same fashion as sys was described. The events concerning each organ are described through its interfaces and a set of messages that organs can exchange with each other. Finally, the functions of sys are decomposed into sub-functions assigned to different organs.

In order for the whole BDM to be coherent, we must check that the behaviors of all the organs put together are consistent with the behavior of sys as described in BDM. Moreover, all those behaviors must be coherent with the functions decompositions. In paragraph 3.2, we will explain how we propose to check these constraints.

3 FOCUS ON SOME FORMALIZATION CHALLENGES

3.1 Requirements on system solicitations
In this section, the intuition behind system solicitations and the corresponding requirements is first presented. Then, we gradually introduce the corresponding mathematical formulation based on the notion of coherence.

3.1.1 Checking the conformance of system’s model with solicitations.
In the process of designing models in a contracting or certification relationship, it is important for the model not only to be structurally valid, but also to describe the system with enough precision that it can be used for analysis. In that regard, contractors must identify precisely the perimeter of the system behavior being modeled. This perimeter is not necessarily complete. It may be desirable in a first design or evaluation stage to only model the nominal behavior of the main features of the system, and not behaviors related to secondary features or error handling. One way to do that is to specify for which stimuli from the system environment the behavior of the system should be described in the corresponding model. In some cases, such as when ignoring behaviors after error messages, those environment stimuli may depend on system’s responses.

In this perspective, the notion of solicitation is introduced. A solicitation specifies a sequence of event exchanges between the system and its environment. Input events (noted e′) specify the stimuli that the environment submits to the system and for which the model must explicitly describe the behavior of the system. Output events (noted e′) specify the reaction/answer of the system that determines the evolution of the rest of the solicitation.

Two categories of requirements on output events can be distinguished. The first category concerns out-events that the system could generate and that are necessary to continue the solicitation.

A typical example of such an out-event is a message sent by the system to notify that a connection or login has been granted and that (in that case only) the rest of the solicitation can be submitted.

The second category of requirements on out-events concerns those events that the system could generate and that prohibit the continuation of its solicitation by the environment. A login error is a typical example of such an event. The corresponding semantics expresses that the solicitation may continue to be submitted to the system if no such events are generated by the system but that no verdict on the conformance of the behavior with the solicitation can be given if such an event is generated.

Formally, we define a solicitation as a sequence of sets of in-events and sets of out-events. These events are within the scope of two distinct operator that indicate whether they are required to appear (the Or operator) or required not to appear (the Not operator) in the model of the system.

Intuitively, a system behavior model Xsys (a set of executions, see 2.3) conforms to a single solicitation s if and only if, starting from the initial state, Xsys fully specifies the behavior of the system after the consumption of any sequence of in-events specified in the solicitation whenever the "conditions" on the occurrence (or not occurrence) of out-events is met.

To verify such a requirement, we introduce the notion of coherence of a solicitation with a set of executions of the system. Without loss of generality, one can consider that an execution reflects a system’s state and can be understood as so in the following definitions.

Informally, coherence of a set of executions X and a solicitation s means that: 1) the direct behavior of the system is specified by its model X when solicited with e′, the first in-event in s; and 2) for any execution x ∈ X that checks e′, any prolongation of x by ⊥-transitions (transitions without trigger) that generates the expected system out-events following e′ in s leads into a state where the coherence of the rest of the solicitation can be checked. In this sense, out-event can be seen as filtering conditions of system executions that need to be checked.

3.1.2 Solicitations with single in-event sequences. For simplicity’s sake, let’s give a first definition of conformance and coherence using solicitations composed of sequences in-events within Or operators only, with the semantics introduced in section 2.2.

Let’s also, for illustration purposes, consider a simple system model consisting in all the paths starting in q0 in the automaton depicted in figure 1 and the solicitations s = Or(a) and s′ = Or(a).Or(b)

![Figure 1: Executions and in-event solicitation sequences](image-url)
Intuitively, one can check that the system’s model $X^{sl}$ conforms to the solicitation $s$ because starting from the initial state $q_0$, $X^{sl}$ specifies the behavior of the system in case it is solicited with $a$, the first $in$-event of $s$. It is worth noting that $X^{sl}$ specifies this behavior including for the case where the system takes the execution path starting with $\bot$ and is in state $q_3$.

However, it is interesting to note that the model $X^{sl}$ does not conform with the solicitation $s'$. Indeed, this model does not specify the behavior of the system when solicited with the $b$ event after it evolves within the execution path starting with $\bot$ then accepts $a$ (thus being in state $q_4$). The model does not specify neither the behavior of the system when it is solicited with $b$ after it evolves with the execution path composed of the event $a$ then $\bot$ (corresponding to state $q_7$).

A first formulation. With this quick example we have outlined the tools we need to formalize conformance between a set of solicitations and a set of executions. Let us consider that we have already examined the beginning of a solicitation containing only $in$-events, and there remains a solicitation $Or(e') \cdot s^5$. Let us also consider that, by exploring the beginning of the solicitation, we have reached a certain set of states, which correspond to a set of executions $X$. We then, to check $Or(e') \cdot s$ we need to verify two things: 1) we have a transition starting with $e'$ from any state of $X$; and 2) $s$ can be checked in any state reachable from the states of $X$, through transitions bearing $e'$, and with any number of $\bot$-transitions afterwards.

We see that in both cases we need to characterize the set of states (that is, of executions) that can be reached from $X$ through transitions bearing $e'$. Therefore we introduce: $Next_{elem}(x, e) = \{ x' \cdot \delta \in X^{sl} \mid in(\delta) = e \}$

To characterize 2), we also need to define the set of states (i.e., executions) reachable starting with a certain set of states and getting through $\bot$-transitions only. This can be defined this way: $Reach_{in}(X) = \{ x' \cdot \delta \cdot x \in X^{sl} \mid x' \cdot \delta \in X \land x = \delta_1 \ldots \delta_n \land \forall i, in(\delta_i) = \bot \}$

We then can write an inference rule for the coherence between $Or(e') \cdot s$ and $X$ this way:

$$\forall x \in X, Next_{elem}(x, e') \neq \emptyset \rightarrow coh_{in}(s, X')$$

$$coh_{in}(Or(e') \cdot s, X')$$

with $X' = Reach_{in}(\cup x \in X Next_{elem}(x, e'))$.

This definition is easily extendable to the case where the $Or$ operator contains a set of possible events instead of a single one. In that case we will have $Next(x, E) = \{ x' \cdot \delta \in X^{sl}[in(\delta) \in E] \}$, and then:

$$\forall x \in X, Next(x, E) \neq \emptyset \rightarrow coh_{in}(s, Reach_{in}(\cup x \in X Next(x, E')))$$

$$coh_{in}(Or(E') \cdot s, X')$$

We also have the following halting cases, which correspond respectively to an empty solicitation (which is coherent with any execution in the system) and an empty set of execution (which indicates an empty set of paths to check):

<table>
<thead>
<tr>
<th>$X \subseteq X^{sl}$</th>
<th>$coh_{in}(s, X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s \neq e \cdot s'$</td>
<td>$coh_{in}(e, X)$</td>
</tr>
<tr>
<td>$s = e \cdot s'$</td>
<td>$coh_{in}(s, s)$</td>
</tr>
</tbody>
</table>

$\bot \neq s$ is freely used to say that $s$ is a solicitation that starts with $e$ and is followed by $s'$. Similarly, $x \cdot \delta$ denotes the execution starting by $x$ and ending with transition $\delta$. We also use $in(\delta)$ to designate the trigger of $\delta$.

Finally, in order to check that a set $S$ of $in$-solicitations conforms to the system behavior (which we will note $Conf(S)$), we need to initialize the $X$ set. At the beginning, the states we consider are those reachable from the initial states with $\bot$-transitions, without matching any $in$-events, that is: $Init_{in} = \{ x \in X^{sl} \mid x = \delta_1 \ldots \delta_n \land \forall i, in(\delta_i) = \bot \}$

We then have $Conf(S) \Rightarrow \forall s \in S, coh_{in}(s, Init_{in})$.

3.1.3 Filtering with output events. The definition of coherence is now extended to take out-events into account. To avoid ambiguous notations, we will note $OrIn$ for $Or$ operators with $in$-events and $OrOut$ for $Or$ operators with out-events. Please recall that out-events are used to determine in which case a solicitation should continue to be submitted to the system. Then, one can understand out-events in a solicitation as filters of the execution paths (or system states) that need to be considered for the remaining solicitation.

For instance, referring to the model depicted in Figure 2, it is easy to verify that this model conforms with the solicitations $s = OrIn(a).OrOut(w, y).OrIn(c, b)$ where $a, b$ and $c$ are $in$ events and $y$ and $w$ are out-events. Indeed, after accepting $a$, the model speci-
an in-event, \( outF(s) = \epsilon \) and \( inF(s) = s \). If \( s \) contains only out-elements, then \( outF(s) = s \) and \( inF(s) = \epsilon \). We will then need to use the \( outF \) part of the solicitation to filter the traversing of \( \perp \)-transitions that we performed in the Reach sets. For this, we must have a way to check whether an execution matches a sequence of out-events. For sets of out-events \( E_1, \ldots, E_n \) and an execution \( x \), we will note \( E_1, \ldots, E_n \perp x \) whenever \( x = \delta_1 \cdot \ldots \cdot \delta_n \) and there exists an increasing function \( \phi : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\} \) such that \( \forall i \in \{1, \ldots, n\}, out(\delta_{\phi(i)}) \cap E_i \neq \emptyset \). The \( \perp \) relation ensures that the elements of the \( E_i \) sets can be found in the output events of \( x \) in the right order.

We consider a solicitation composed only of out-events. We then define a relation \( check \) by the following inferences:

\[
\begin{align*}
\text{CheckAs} & : \text{check}(x, \epsilon) \\
\text{CheckOr} & : E_1, \ldots, E_n \perp x_1 \\
& \cdot \text{check}(x_1 \cdot x_2, OrOut(E_1) \cdot \ldots \cdot OrOut(E_n) \cdot s)
\end{align*}
\]

We see that this relation checks whether the \( OrOut \) solicitations appear in some prefix of the execution we consider. We can easily extend this relation for the Not solicitations. The simplest case is when the solicitation contains only occurrences of Not. Then we must check that no prefix of \( x \) match those events:

\[
\begin{align*}
\text{CheckNot} & : \forall x_1, x_2 = x, E_1, \ldots, E_n \not\perp x_1 \\
& \Rightarrow \text{check}(x, Not(E_1) \cdot \ldots \cdot Not(E_n))
\end{align*}
\]

The last case is a bit more complicated. When a series of Not is followed by \( OrOut \), then events in the scope of the Nots must never be seen before an occurrence of the event in \( OrOut \) and at least one of these occurrences must exist:

\[
\begin{align*}
\text{CheckNotOr} & : \forall x_1, x_2 = x, E_1, \ldots, E_n \not\perp x_1 \\
& \exists x_1, x_2 = x, \text{check}(x_2, OrOut(E_{n+1} \cdot s)) \\
& \Rightarrow \text{check}(x, Not(E_1) \cdot \ldots \cdot Not(E_n) \cdot OrOut(E_{n+1} \cdot s))
\end{align*}
\]

With the \( check \) relation fully defined, we can extend our previous sets using it. We want to define the set of states we can reach, not only by traversing \( \perp \)-transitions, but also by checking that those transitions match a series of out-events. We have then this new definition:

\[
\text{Reach}(X, s) = \left\{ \begin{array}{l}
\exists x' \cdot \delta \in X \\
x' = \delta_1 \cdot \ldots \cdot \delta_n \\
\forall i, in(\delta_i) = \perp \\
\text{check}(\delta \cdot x, s)
\end{array} \right\}
\]

We can then extend our definition of coherence:

\[
\text{Conf}(s) \iff \forall s \in S, coh(inF(s), \text{Reach}(\cup_{x \in X} Next(x, E^i)), outF(s)))
\]

We also need to update the set of initial states with our checking of out-events prefixes: \( Init(s) = \{ x \in X^{sys} \mid x = \delta_1 \cdot \ldots \cdot \delta_n \land \\
\forall i, in(\delta_i) = \perp \land \text{check}(x, s) \} \)

And lastly we have:

\[
\text{Conf}(S) \iff \forall s \in S, coh(inF(s), Init(out(s)))
\]

Finally, we have written a formal relation that checks whether the set of executions \( X^{sys} \) which models the system’s behavior covers the set of solicitations of interest for the BDM.

### 3.2 Requirements on organic decomposition

In this section, modeling requirements specifically related to the organic decomposition of the system are introduced. An informal presentation of these requirements is first given. Then, we present the necessary data structures and algorithms to support reasoning on behavioral decomposition.

#### 3.2.1 Approach

Our aim with these requirements is to check that the behavior of all the organs of the models, composed together, is similar to the behavior of the whole system.

The behavior of each organ (as well as the whole system) is described with a set of executions \( X \). Each execution \( x \in X \) is a finite sequence of transitions \( \delta \) (the ending state of each being the starting state of the next one). Each transition \( \delta \) is of the form \(\delta = x \xrightarrow{e/F} x'\) where \( x \) and \( x' \) are the source and target states, \( e \) is a trigger event (which should be an input event, marked with \( in \)), \( F \) is a set of functions triggered by \( e \) and \( E \) is a set of effect events (which should be output events, marked with \( out \)). Remember that we do not want to prejudice the order between the elements of both \( E \) and \( F \) (this may be unknown or intentionally unspecified).

Input/Output Automata [39] are used to address similar problems [16]. However, our sets of executions are strictly more expressive than those extracted from Input/Output Automata (see Sect. 2.3) and their solutions are not directly applicable to our case. Therefore, we introduced a slightly different form of parallel composition in which messages between organs are buffered. This buffer does not need to be local to each organ, because our events are identified with interfaces, thus a given event could only be happening at a specific organ. Consequently, we just use a pool of messages as a buffer. Initially, this pool will be filled with an outside event which triggers the whole organic execution. Then, each transition consists in picking an event in the pool, transitioning any organ
that can transition with this event as a trigger, and then updating the pool with all output events sent by those organs.

3.2.2 Illustrative example. To explain our method, we will use the example presented in Figure 3 which represent both the system’s organs and their behavior. The system is composed of three organs, noted IO, A and B. On the black-box level, the system has a single transition behavior of the following form:

\[ \delta_{\text{sys}} = q \mapsto (\text{msg},s)/\phi)/\text{(ok},s) \rightarrow q' \]

Each organ also has a behavior, described in Figure 3. From this description, we can infer the general behavior that is supposed to happen while executing the \( \phi \) function:

1. The environment sends \( \text{msg} \) to the system, on the interface \( s \) which is actually attached to the organ IO.
2. IO enters state \( Q_1 \) and sends \( \text{msg} \) on both the \( \alpha \) and \( \beta \) interfaces.
3. \( \text{msg} \) is received by \( A \) on \( \alpha \) and by \( B \) on \( \beta \).
4. Both \( A \) and \( B \) executes their internal function (respectively \( f \) and \( g \)) and each send an \( \text{ack} \) message back to IO.
5. Once IO has received \( \text{ack}_A \), it enters \( Q_2 \) and waits for \( \text{ack}_B \).
6. Once IO has received \( \text{ack}_B \), it sends out \( \text{ok} \) and gets back to its initial state.

We will show how our method allows to check that the organic behavior is a correct decomposition relatively to the execution \( \delta_{\text{sys}} \).

3.2.3 Multi-transitions. We acknowledge that our model gives us the connections between organs. In this context, this means that given an output event \( e \), we can get the corresponding input event \( \tilde{e} \) that can receive this message. We note \( \tilde{E} = \{ \tilde{e} \mid e \in E \} \). In our example, we have for instance \( (\text{msg},a) = (\text{msg},a) \) because \( a \) and \( \alpha \) are two interfaces talking to each other.

Moreover we note \( \text{Events} \) the set of events appearing between two organs of our model. For instance, in our example, \( (\text{msg},a) \in \text{Events} \) but \( (\text{msg},s) \notin \text{Events} \) because \( s \) is an outer interface.

As we deal with executions instead of automata, we will use prefixes of executions instead of states\(^5\). We consider a set of \( n \) organs. Each organ has a behavior described by a set of executions \( X_i \).

In the case where executions are described by an automaton, states of this automaton can be used instead of prefixes.

Let \( Q = \{X_1, \ldots, X_n\} \) and \( Q' = \{X'_1, \ldots, X'_n\} \) be elements of \( \text{pref}(X_1) \times \ldots \times \text{pref}(X_n) \). We note \( Q \xrightarrow{e/F/E} Q' \) the relation defined by:

\[ Q \xrightarrow{e/F/E} Q' \iff \exists i \in \{1, \ldots, n\}, X_i \xrightarrow{e/F/E} X'_i \]

In our example, the prefixes are the set of states. All the possible sets are \( (Q_0, Q_3, Q_4) \), \( (Q_1, Q_3, Q_4) \) and \( (Q_2, Q_3, Q_4) \). According to our previous definition, we have \( Q \xrightarrow{e/F/E} Q' \) if one of the states can change state with the given transition, and the other states do not move. Thus, the following transitions are possible:

\[
\begin{align*}
(Q_0, Q_3, Q_4) & \xrightarrow{(\text{msg},a)/f(1))/\text{(ack}_A,a) \rightarrow (Q_0, Q_3, Q_4) \\
(Q_0, Q_3, Q_4) & \xrightarrow{(\text{msg},s)/\emptyset)/\text{(ack}_A,a), (\text{msg},\beta) \rightarrow (Q_1, Q_3, Q_4)
\end{align*}
\]

In the following, we note \( Q_e = (e, \ldots, e) \) the empty multi-prefix. In the case where we consider states in an automaton, \( Q_e \) would be any set of initial states. In our example, \( Q_e = (Q_0, Q_3, Q_4) \).

We then define a multi-transition \( \delta = (Q, E) \xrightarrow{e/F/E} (Q', E') \) whenever there exist two multi-sets of events \( E_{\text{new}} \) and \( E_{\text{org}} \) such that we have:

\[
\begin{align*}
e & \notin \bot \Rightarrow e \in E \\
Q & \xrightarrow{e/F/E} Q' \Rightarrow Q' \\
E_{\text{new}} & = E_{\text{new}} \cap \text{Events} \\
E' & = (E_{\text{new}} \cup E_{\text{org}}) \cup (E_{\text{org}} \setminus E_{\text{new}})
\end{align*}
\]

In this transition, we ensure that (1) the triggering event is either the empty event or is in the buffer-pool; (2) we advance an organ according to the triggering event, it produces a set of events \( E_{\text{new}} \), (3) we select the events that are transferred between the organs, (4) the buffer-pool is updated: we remove\(^{10}\) the triggering event, we add the input events corresponding to all the outputs between organs and finally we add the events that are outside the organs (in order to check them later with the system behavior).

Let us build such a multi-transition with our example. If we take \( Q = (Q_0, Q_3, Q_4) \) and \( E = \{(\text{msg},a), (\text{msg},b)\} \), then we can build a transition with \( e = (\text{msg},a) \) in order to fulfill condition (1). We then have a transition fulfilling (2) with \( F_{\text{new}} = \{f(1)\} \) and \( E_{\text{new}} = \{(\text{ack}_A,a)\} \). As \( (\text{ack}_A,a) \in \text{Events} \), we have \( E_{\text{org}} = E_{\text{new}} = \{(\text{ack}_A,a)\} \) according to (3). Then \( \text{(ack}_A,a) = (\text{ack}_A,a) \) and thus \( E_{\text{org}} = \{(\text{ack}_A,a)\} \). As \( (E_{\text{new}}\{\text{e}\}) = \{(\text{msg},b)\} \) and \( (E_{\text{org}}\setminus E_{\text{new}}) = \emptyset \), we have \( E' = \{(\text{ack}_A,a), (\text{msg},b)\} \). Finally, we have the following multi-transition:

\[
((Q_0, Q_3, Q_4), \{(\text{msg},a), (\text{msg},b)\}) \xrightarrow{(\text{msg},a)/f(1)} ((Q_0, Q_3, Q_4), \{(\text{ack}_A,a), (\text{msg},b)\})
\]

3.2.4 Multi-executions. A multi-execution is an execution composed with multi-transitions. We note \( \text{MExec}(X_1, \ldots, X_n) \) the set of multi-executions constructed on the executions \( X_1, \ldots, X_n \). Moreover, we note:

\[
\begin{align*}
\text{MExec}(Q, E) & \rightarrow (X_1, \ldots, X_n) \text{ the set of multi-executions starting with } (Q, E) \\
\text{MExec}_{\rightarrow}(Q, E) & \rightarrow (X_1, \ldots, X_n) \text{ the set of multi-executions ending on } (Q, E) \\
\text{MExec}(Q, E) & \rightarrow (Q', E') \rightarrow (X_1, \ldots, X_n) \text{ the set of multi-executions starting with } (Q, E) \text{ and ending on } (Q', E').
\end{align*}
\]

---

\(^5\)The same state can be in several executions with different predecessors and successors, without the ability to "jump" from one execution to the other.

\(^{10}\)We remove one occurrence only since the pools are multi-sets.
This is a multi-execution of our example:

\[
((Q_0, Q_0, Q_1), \{(msg, s)\}) \quad (msg, s)/\emptyset
\]
\[
((Q_0, Q_0, Q_1), \{(msg, a), (msg, b)\}) \quad (msg, a)/\{f\}
\]
\[
((Q_1, Q_0, Q_1), \{(ack_A, a), (msg, b)\}) \quad (msg, b)/\{g\}
\]
\[
((Q_1, Q_0, Q_1), \{(ack_A, a), (ack_B, b)\}) \quad (ack_A, a)/\emptyset
\]
\[
((Q_2, Q_1, Q_1), \{(ack_B, b)\}) \quad (ack_B, b)/\emptyset
\]
\[
((Q_0, Q_0, Q_1), \{(ok, s)\})
\]

We then have, in this example, a multi-execution \(u\) which, with \(\bar{X} = (X_{in}, X_A, X_B)\), belongs to:

\[\text{MExec}_{\mathfrak{C}}((Q_0, Q_0, Q_1), \{(msg, s)\}) \rightarrow ((Q_0, Q_0, Q_1), \{(ok, s)\})\]

3.2.5 Refinement. Given a multi-set \(E\), we note \(E_{in}\) the multi-set of elements of \(E\) which are system input events and \(E_{out}\) the multi-set of elements of \(E\) which are system output events.

With those objects, we define the notion of refinement between organic composition and system execution. We say that a multi-execution \(u\) is a refinement of a system transition \(\delta = q \xrightarrow{e/f_{\text{new}}/f_{\text{new}}} q'\) along \(((Q, E), (Q', E'))\) and we note \(\text{refine}_{(Q, E)} \rightarrow (Q', E') (\delta, u)\) if we have:

\[
e \neq \bot \Rightarrow \\
u \in \text{MExec}_{\mathfrak{C}}((Q, E), (Q', E')) (X_1, \ldots, X_n) \land e \notin E' \\
e = \bot \Rightarrow u \in \text{MExec}_{\mathfrak{C}}((Q, E), (Q', E')) (X_1, \ldots, X_n) \\
E'_{\text{out}} = E_{\text{new}}
\]

Considering \(\delta_{\text{sys}} = q \xrightarrow{\{msg, s\}/\{φ\}/\{ok, s\}} q'\) and our multi-execution \(u\), we can have the above requirements if we take \(Q = Q' = (Q_0, Q_0, Q_1), E = \emptyset\) and \(E' = \{(ok, s)\}\), thus we have:

\[\text{refine}_{(Q_0, Q_0, Q_1), \emptyset} \rightarrow (Q_0, Q_0, Q_1), \{(ok, s)\}) (\delta_{\text{sys}}, u)\]

Given a system execution \(x = \delta_1 \cdots \delta_n\), a sequence of multi-executions \((u_1, \ldots, u_n)\) refines \(x\), noted \(\text{refine}(x; (u_1, \ldots, u_n))\), if there exist multi-sets \(E_0, \ldots, E_n\) and prefixes \(\hat{Q}_0, \ldots, \hat{Q}_n\) belonging to \(\text{pref}(X_1) \times \cdots \times \text{pref}(X_n)\) such that:

\[
E_0 = \emptyset \\
Q_0 = \hat{Q}_e \\
\forall i > 0, \text{refine}_{(Q_{i-1}, E_{i-1})} \rightarrow (Q_i, E_i) (\delta_i, u_i)
\]

In our example, if \(x\) is the execution consisting in only one occurrence of \(\delta_{\text{sys}}\), then we have \(\text{refine}(x, u)\). This is possible because \((Q_0, Q_0, Q_1) \in \hat{Q}_e\).

3.2.6 Requirements. With those structures, we can state the following requirements:

Any system execution has an organic equivalent

To check this, we must ensure that for any system execution \(x\), there are some multi-executions \(u_1, \ldots, u_n\) belonging to \(\text{MExec}(X_1, \ldots, X_n)\) such that we have \(\text{refine}(x, (u_1, \ldots, u_n))\).

Any emerging behavior has a system equivalent

Although some organic behavior can be invisible at the black box level (and in this case, we permit that it was not specified in the system behavior), we have to ensure that the composition of organic behaviors does not introduce a new way to react to outside stimuli. To do that, we check that whenever a multi-execution is extended, there already is a system execution corresponding to this extension. The formal description of this requirement is out of the scope of this paper.

This section presented only two generic soundness requirements. BDMs include additional soundness requirements related to other sub-models which are out of the scope of this paper due to lack of space. BDMs also allow to express more domain-specific completeness requirements through solicitations, such as, for embedded systems, requiring to explicit the behavior of the system depending on the energy sources available. In complement, BDMs related to embedded systems would often include a functional view (Sect. 2.4) focused on the events related to variations in the energy sources provided to the system.

4 RELATED WORKS

Models’ coherence verification is not a new challenge. Indeed, there are many existing approaches to address this problem. In this section we only report the works that, like our own, are focused on the process of modeling and verification of the particular category of properties that are inherent to the models regardless of the modeled system.

A noticeable fact when studying techniques that address these challenges is that most of them focus on verifying UML/OCL models [28]. This is, however, no surprise since UML as well as OCL are de facto standards for systems’ modeling. These approaches can be distinguished according to different points of interest, for instance, the properties covered. These approaches can cover satisfiability properties [3, 4, 8, 9, 12, 38, 41], constraints-related properties [22, 23] or both [11, 27, 46–48, 54].

These techniques can also be distinguished according to the extent to which they support the OCL language. Specifically, we can identify a first category of approaches that support the full OCL language [3, 4, 8, 9, 11, 24–27, 47, 48, 54, 56, 57], a second category that supports a subset of OCL [13, 46, 49] and a third one that provides no support for OCL [12, 22, 23, 38, 41]. We can also distinguish these approaches with regard to the underlying verification techniques. For instance, approaches can rely on CSP [11, 12, 27], Description logics [12, 46], Relational logics [3, 4, 24–26], HOL [2, 8, 9, 49], process algebras [42, 52], etc. Other criteria can be considered such as tool support, assistance in correction of detected problems, termination guarantee.

It is worth noting that other works deal with the verification of validity requirements on other types of models. For instance, Weckesser et al. address the problem of checking the consistency of Clafier models [59, 60], Bertram et al. check the consistency of Component and Connectors view models [6, 40] and Schrefl et al. verify the consistency of behavior diagrams [55]. An important aspect common to all the aforementioned approaches is that the proposed verification techniques are closely correlated to the modeling language (UML/OCL, Clafier, C&C views and behavior diagrams). On the contrary, the work described in this paper is independent of
any particular modeling language thanks to the agnostic approach used to introduce the semantic domain of BDMs. Still, this work is applicable to a variety of modeling formalisms by mapping the conceptual elements of these formalisms to BDM.

Another important difference with these techniques is that they all (except for [55]) focus on static models while our interest also includes the behavioral aspects of the system. This particular concern has been addressed in another family of works. Specifically, most related to our own description of behavioral decomposition are the works on Interface and Input/Output Automata [16, 39]. Interface Automata provide formal technical tools to express and check the compatibility of the components of a given system and to verify that their combination actually produces the overall desired behavior. This is achieved by providing a coherent theory consisting in a semantic domain, a refinement operator and an equivalence relation. The base formalism has been extended to address various issues such as error propagation in system refinements [21], compositional verification [64] or requirements traceability [18]. Although we partially share with this family of techniques some of the objectives, the behavioral decomposition sub models, the particular semantics of synchronization mechanism adopted in the case of our semantic domain and the distinction between must and may transitions in interface theories make their exploitation in our approach somehow cumbersome.

Finally, although only outlined in this article, functional views are an important part of the work on BDMs. It is worth noting that we understand views in our current work as slices of the system. In this sense, our work is comparable to previous ones [1, 10, 34, 45, 50]. However, a notable difference with these approaches is that our interest is not slices generation, slices update, nor generation of slices that are syntactically valid with respect to the original meta-models. Our requirements concern the semantic conformance of the functional views/slices with the original global model of the system.

5 CONCLUSION AND FUTURE WORKS

In this paper, we proposed a mathematical framework to design agnostic models called Behavior Decomposition Model. We showed how the submodels of a BDM can be used to describe the black-box behavior of a system and its first-level decomposition. Our approach provides two algorithms to check some internal coherence relation. The base formalism has been extended to address various issues such as error propagation in system refinements [21], compositional verification [64] or requirements traceability [18]. Although we partially share with this family of techniques some of the objectives, the behavioral decomposition sub models, the particular semantics of synchronization mechanism adopted in the case of our semantic domain and the distinction between must and may transitions in interface theories make their exploitation in our approach somehow cumbersome.

Finally, although only outlined in this article, functional views are an important part of the work on BDMs. It is worth noting that we understand views in our current work as slices of the system. In this sense, our work is comparable to previous ones [1, 10, 34, 45, 50]. However, a notable difference with these approaches is that our interest is not slices generation, slices update, nor generation of slices that are syntactically valid with respect to the original meta-models. Our requirements concern the semantic conformance of the functional views/slices with the original global model of the system.

ACKNOWLEDGMENTS

This work was in part conducted at Capgemini Rennes, France, and benefited from discussions with other members of the team including Cédric Lalellec, Gidlas Preniel-Cabic, Evelyne Kabore and team leader Hélène François. The authors are also grateful to the reviewers for their helpful comments.

REFERENCES

[20] Sascha Fendrich and Gerald Lüttgen. 2016. A generalised theory of interface and team leader Hélène François. The authors are also grateful to the reviewers for their helpful comments.

https://doi.org/10.1016/0304-3975(95)92307-J


Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.

https://doi.org/10.1016/0304-3975(95)02307-J

Meeting on the mathematical foundation of programing semantics.