# Modeling and Enactment Support For Early Detection of Inconsistencies In Engineering Processes

István Dávid*†, Bart Meyers*†, Ken Vanherpen*†, Yentl Van Tendeloo*, Kristof Berx†, Hans Vangheluwe*†‡
*University of Antwerp, Belgium
{istvan.david, bart.meyers, ken.vanherpen, yentl.vantendeloo, hans.vangheluwe}@uantwerp.be
†Flanders Make, Belgium
kristof.berx@flandersmake.be
‡McGill University, Montréal, Canada

***Abstract*—Managing inconsistencies between models is a key challenge in engineering processes of complex systems. Early detection of inconsistencies results in more efficient processes, because it can reduce the amount of re-execution of costly engineering activities.**

**In this paper, we propose an approach for early inconsistency detection in engineering processes. In our approach, the engineering process is explicitly modeled, along with the important characteristics and constraints of the system, imposed by the requirements and system specifications. This information is then used to enact the process and augment it with a run-time consistency monitoring service. Inconsistencies are expressed as a satisfiability problem of the constraints. Early detection of inconsistencies is achieved by monitoring the constraints, that is, checking their satisfiability at specific points of the process. Our approach is supported with a framework which includes a visual process modeling tool, a process enactment engine and a state-of-the-art symbolic solver for early inconsistency detection.**

***Index Terms*—inconsistency management, process modeling, multi-paradigm modeling, cyber-physical systems, mechatronics**

## I. Introduction

Engineering complex heterogeneous systems requires a collaboration between stakeholders of various domains. These stakeholders have different views [1] on the system and reason only about the system characteristics relevant to their respective views. Some characteristics of the system, however, cannot be related to only one view or domain. Such cases result in overlaps between specific views and give rise to inconsistencies between those views. For example, selecting a battery for an autonomous vehicle will have an impact to the mechanical view of the system (which reasons about the mass of the battery), and to the electrical view (which reasons about the capacity of the battery).

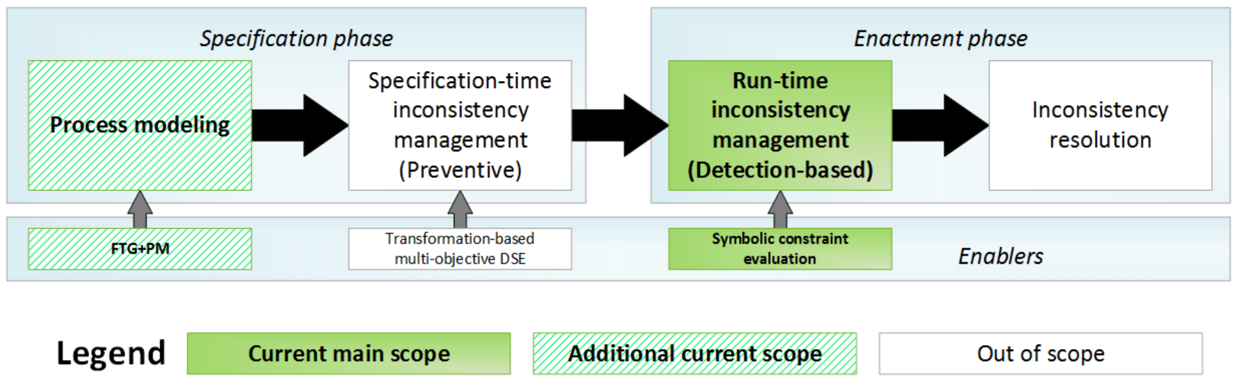
As proposed by Finkelstein et al [2], instead of simply just removing inconsistencies from the design, we need to think about "managing consistency". One of the core activities of inconsistency management is the detection of inconsistencies. It is preferred that detection is achieved as early as possible, because the earlier the inconsistencies are detected, the lower the costs of resolving them are. *Early detection of inconsistencies* also provides more freedom in choosing the appropriate resolution and tolerance techniques [3] [4].

To support the understanding of the origins and root causes of inconsistencies, we reuse one of the main guidelines of multi-paradigm modeling (MPM) [5] and we place the *process* manipulating the models of the system into the center of our work. Inconsistencies are defined, detected and analyzed with respect to the process.

### *Inconsistencies*

The presence of a process enables richer semantics to reason about the various types of inconsistencies. The traditional interpretation of inconsistencies comes from the area of multi-view modeling, where the notion of the process is not necessarily present, e.g. in [6]. In such settings, inconsistencies are caused by multiple stakeholders modifying the system design in a *parallel* fashion and introducing discrepancies between the views of the system, in specific attributes, values, properties. The strong notion of a process, however, allows to reason about more "stateful" types of inconsistencies, i.e. the ones that are caused by a *sequence* of modifications. Discrepancies are not necessarily introduced between various views, but typically in constraints regarding the system, as the design decisions shape the system more and more specific.

In our previous work [7], we proposed the foundations of a formalism to reason about inconsistencies in the presence of a strongly typed process model, based on the FTG+PM formalism [8]. The formalism enables modeling system characteristics as (ontological and linguistic) *properties* in conjunction with the engineering *process*. The process is analyzed and various management patterns are applied to *prevent* potential inconsistencies to occur. That is, the process is transformed in a way that no inconsistency remain unnoticed when the process is enacted. (Referred as *Specification-time inconsistency management* in Figure 1.) In these cases, one typically addresses inconsistencies due to the parallel activities.



Specification phase
Process modeling
Specification-time inconsistency management (Preventive)
Enactment phase
Run-time inconsistency management (Detection-based)
Inconsistency resolution
FTG+PM
Transformation-based multi-objective DSE
Symbolic constraint evaluation
Enablers
Legend
Current main scope
Additional current scope
Out of scope

Figure 1: The main focus areas of our research, with the scope of the current paper (*Run-time inconsistency management*) highlighted. For this, we revise the previously addressed formalism for *Process modeling*. *Specification-time inconsistency management* is covered in [7]. *Inconsistency resolution* is a future work.

### *Scope of the paper*

In this paper, we focus on *early* detection of inconsistencies during the *enactment phase* of the engineering processes, i.e. at run-time (Figure 1). Such scenarios necessitate more precise modeling of the properties of the system with respect to the engineering process. To this end, we revise our original process modeling formalism and introduce the notion of attributes (in models) and capabilities (in formalisms) in the FTG+PM. Additionally, we provide a state-of-the-art symbolic solver for detecting inconsistencies in actual, real modeling artifacts. The contributions are presented through an industry level example of a complex mechatronic system, an automated guided vehicle (AGV), coming from one of our partners.

The core contribution of this paper is a methodology for explicit modeling of the characteristic *attributes* of the system and allow defining *constraints* upon them to express inconsistency rules in order to monitor the process for inconsistencies at runtime. The process is augmented with consistency checks during the enactment, which can be viewed as special activities of the process. These activities are not explicitly modeled and do not carry relevant information from the engineering point of view and therefore, they remain hidden.

The explicit modeling of attributes and constraints means that the information relevant to inconsistency management is being conceptually “lifted” from the models containing those attributes and constraints. (Although, they are still present in the models themselves.) This modeling step may be expensive for larger processes, but it has to be done only once for a process. Attributes and constraints operate on multiple meta-levels, which enables reasoning about *capabilities* of certain modeling formalisms. This, due to the strongly typed process model, provides additional vital information regarding potential inconsistencies along the process. This combination of modeling paradigms (i.e. multi-level, multi-abstraction, process-oriented) is novel in the state-of-the-art.

To support our ideas, we provide a prototype tool[1], which supports (i) specification of the above mentioned aspects of the engineering process, and (ii) execution of the modeled process in a managed way, i.e. by monitoring the (in)consistency of the modeling artifacts.

[1]Available from: http://istvandavid.com/icm.

The rest of the document is structured as follows. In Section II, we present a motivating example of the engineering of a complex heterogeneous system. Using the example, we propose a modeling formalism in Section III to capture various system characteristics and constraints, which serve as (in)consistency rules for the process. In Section IV, we present the enactment aspects of the previously specified process, including (i) the execution of the process additionally supported with tool interoperability, and (ii) the monitoring and detection of inconsistencies. Finally, we conclude our work by giving an overview on the state of the art in Section V and by wrapping up the paper in Section VI.

## II. MOTIVATING EXAMPLE

To motivate our work, we use a case study of the design of an AGV. The goal of the AGV is to carry out a mission of transporting a payload on a specific trajectory between two locations. Being a complex mechatronic system, the requirements of the AGV are refined into specifications by stakeholders of the different involved domains, such as (i) mechanical specifications: sufficient room on the vehicle to place payload; (ii) control specifications: following the defined trajectory with a given maximal tracking error; (iii) electrical specifications: autonomous behavior, defined as the number of times that it needs to be able to perform the movement before needing to recharge; (iv) product quality specifications: the previous specifications should be achieved at a minimal cost. The design process needs to determine the sizing of the different components (motors, battery, platform) and tune the controller. The process requires a collaboration between different stakeholders and their domain-specific engineering tools, such as CAD tools for platform design, Simulink and Virtual.Lab Motion for multi-body simulations, AMESim for multi-physical simulations during drivetrain design. The motor and battery selection activities use databases maintained in Excel files. Since these tools work with different modeling formalisms, reasoning over the consistency of the system as a whole properties poses a complex problem. By explicitly modeling attributes and constraints of the system and associating them with the engineering activities, the engineering process can be augmented with automated consistency monitoring.

*Running example*

The total mass of the AGV ($m_T$) is a sum of the mass of the battery ($m_B$), the mass of the motor ($m_M$) and the mass of the platform ($m_P$):

$$m_T = m_P + m_M + m_B. \tag{1}$$

During the engineering process, and specifically: during the requirements analysis, constraints are applied on the attributes:

$$\begin{aligned} m_T &\leq 150\ [kg], \\ m_P &\leq 100\ [kg], \\ m_M &\leq 50\ [kg], \\ m_B &\leq 10\ [kg]. \end{aligned} \tag{2}$$

These constraints must be respected throughout the whole process, otherwise the model of the system becomes inconsistent.

Additionally, all the masses must be positive numbers, as constrained by the laws of physics.

$$mass > 0\ [kg]. \tag{3}$$

Obviously, the notion of masses specific to the system, i.e. $m_T$, $m_B$, $m_M$ and $m_P$, are of a different nature than the general $mass$ concept, as $mass$ is not related to a specific part of the system, but is more abstract. The concept of $mass$, in fact, can be viewed as *type* to the system-specific masses: $m_T$, $m_B$, $m_M$ and $m_P$ are all masses and therefore, a constraint on $mass$ imposes a constraint on each system-specific mass. This means the constraint in Equation 3 must hold for each system-specific mass.

*Inconsistencies.* An inconsistency can occur, for example, in the following scenario.

**Step 1:** A platform is selected with a mass of 100 kg. ($m_P = 100\ [kg]$)
**Step 2:** A motor is selected with a mass of 50 kg. ($m_M = 50\ [kg]$)
**Step 3:** A battery is selected with a mass of 10 kg. ($m_B = 10\ [kg]$)

At this point, an inconsistency can be detected. Even though the selected components satisfy their respective constraints imposed in Equation 2, the total mass now becomes 160 kgs (due to Equation 1), which leads to a violation of a constraint in Equation 2 and therefore: the design is considered inconsistent.

Factoring in Equation 3, however, allows an earlier detection of inconsistencies. Already in *Step 2*, Equation 1 can be rewritten as follows:

$$m_T = 150 + m_B.$$

Since Equation 3 holds for any mass, and consequently for $m_B$ as well, it can be inferred that after selecting a battery (and thus filling in $m_B$ in this equation), the total mass will be greater than 150, thus violating the constraint in Equation 2.

Such an *early* detection of inconsistencies may save significant costs in the specific engineering process, because it reduces the amount of iterations over complex engineering activities if the design is detected to be inconsistent. Early detection of inconsistencies requires (i) reasoning over constraints on different meta-levels; and (ii) efficient constraint solving algorithms. In Section III, we provide a formalism for the former requirement, and in Section IV we discuss the latter requirement.

## III. Modeling system characteristics

In this section, we present a formalism (i.e. a language with semantics) to model the engineering process and its consistency constraints discussed in Section II. The formalism builds on the FTG+PM formalism [8], and its foundations have been presented in our previous work [7].

### A. FTG+PM*: A brief overview*

The FTG+PM formalism enables the usage of process models (PM) in conjunction with the model of formalisms and the transformations between those (the formalism-transformation graph - FTG). As shown in Figure 2, *Formalisms* and *Transformations* serve as a type system to the *Objects (models)* and the *Activities* of the process, respectively.

The strong type system of the formalism fits well with the problem sketched in Section II as it supports reasoning on different meta-levels. To enable modeling the problem at hand, we introduce new elements to the formalism. In the following, we elaborate on these new elements in greater details.

As shown in Figure 2, the original FTG+PM metamodel is extended by a third typing relationship: between *Capabilities* and *Attributes*. Furthermore, *Constraints* can be defined for both of these elements to capture consistent states of the models in the process. These added elements are the foundations to the inconsistency monitoring approach we are about to present.

### *B. Attributes and capabilities*

Attributes represent characteristic values of the system. These values can be persisted in an object (of a model) and queried directly; or can be derived by a complex query.

*Definition 1 (Attribute):* An attribute is defined as a result of a query over a model, which query is composed of (potentially multiple instances of) projection and aggregation operations.

In the example in Section II, the mass of the platform ($m_P$), the mass of the battery ($m_B$) and the mass of the motor ($m_M$), are attributes that are directly persisted in a mechanical model, thus are directly queried from the mechanical model (each by a respective projection); while the total mass ($m_T$) is an aggregation of the previous three masses and is not necessarily persisted in the model. (We assume the time required for

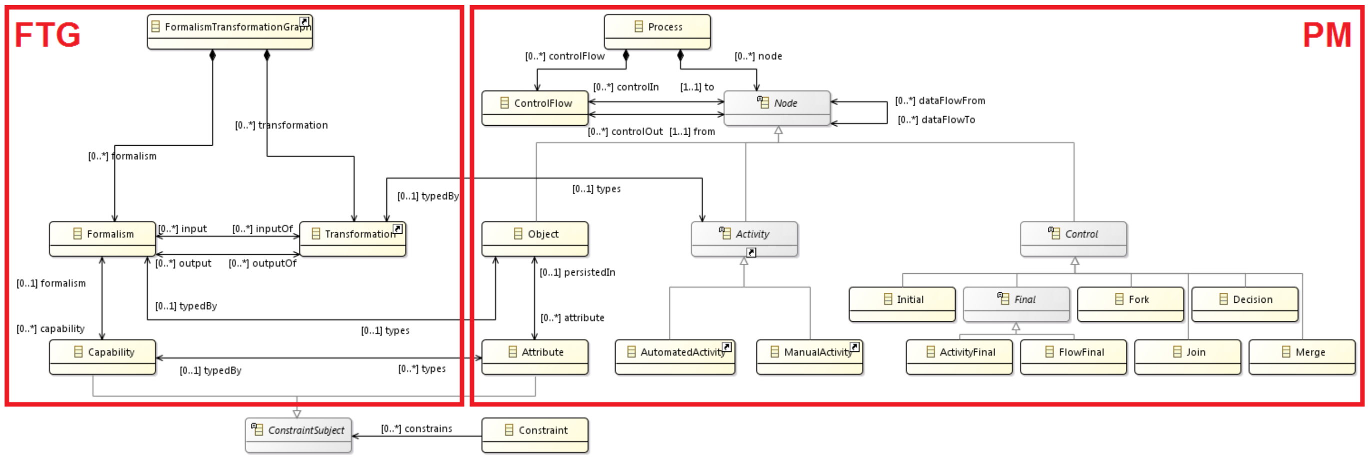


Figure 2: Excerpt from the extended FTG+PM metamodel used in our work.

executing the query being negligible compared to the length of a real life engineering process.) After obtaining $m_B$, $m_M$ and $m_P$, $m_T$ is obtained as the *sum* aggregation of the former three attributes, as shown in Equation 1. Consequently, as shown in Figure 2, attributes are situated on the PM side of the FTG+PM, i.e. on the instance level.

As discussed in Section II, the concept of $mass$ is different from the concept of the masses specific to the system: $m_T, m_B, m_M$ and $m_P$ are related to the notion of $mass$ by a typing relationship. In our framework, we call these meta-attributes capabilities.

*Definition 2 (Capability):* A capability of a formalism expresses the ability of a model, corresponding to the formalism, to reason about attributes a set of system characteristics.

In the running example, Matlab is used for defining the (simplified) mechanical model of the AGV. The Matlab language, in this sense, is able to reason about masses. (Although, mass-like attributes are just ordinary data structures from Matlab's point of view.)

### C. Constraints

To make use of attributes and capabilities for consistency management purposes, constraints are imposed on these to define consistent states of the engineering process.

*Definition 3 (Constraint):* A constraint defines the desired characteristics of the system, i.e. it is a selection of an interval over the domain of the previously obtained attribute.

In a typical engineering process, algebraic (Equation 1), arithmetical (Equations 2 and 3) and logical formulas are used as constraints. As shown in Figure 2, constraints can be applied on both the PM and the FTG side.

*Definition 4 (Consistent design):* The design of the system is considered to be consistent at a given point of the process iff there are no violated constraints.

The detection of the violated constraints is discussed in Section IV.

### D. Modeling the example

After defining the core concepts, we use our prototype tool to model the attributes, capabilities and constraints of the engineering process. The tool provides a visual interface for modeling. It was built on top of the Eclipse platform, implemented using the Sirius framework [10], and it is available as an open-source software.

#### Attributes and their constraints

Figure 3 shows an excerpt from the full model of the example, with the attributes of the running example and their constraints modeled.

Attributes are denoted by light red rectangles, and constraints by darker red rectangles. There are four attributes in the figure, one for each of the masses in Section II. Apart from the *totalMass*, the three other masses are persisted in the *mechanicalModel*, as shown by the prefix in the names of the attributes. The total mass is not persisted in the mechanical model, but it is a result of an aggregation of the other three masses (Equation 1). This equation is captured in the rightmost constraint, as shown by the formula. The other four constraints correspond to the four sub-equations of Equation 2.

The header of the constraint contains its level of precision. In this example, all of the constraints are of level $\mathbb{L}3$. As defined in our previous work [13], the level of precision reflects what information a constraints carries:

- $\mathbb{L}1$: the **fact of influence** is known, its extent is not;
- $\mathbb{L}2$: **sensitivity information** between two values is expressed, e.g. by Forrester system dynamics [14];
- $\mathbb{L}3$: the constraint can be expressed using an exact **mathematical relationship**.

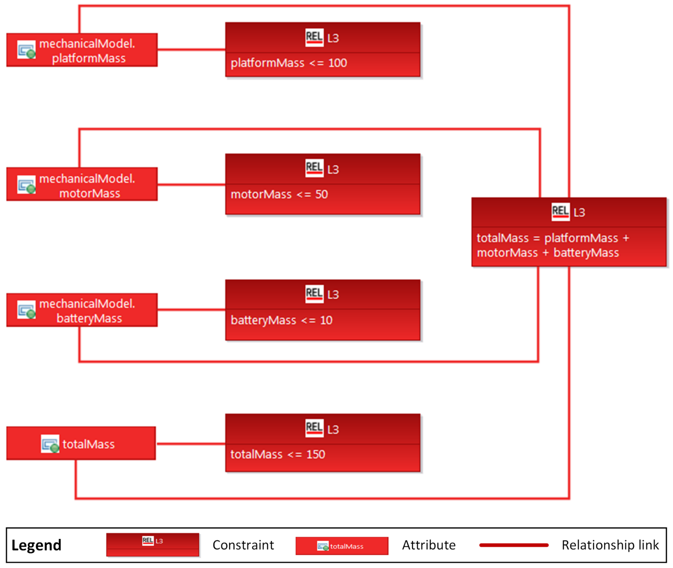


Figure 3: Attributes and constraints.

In this work, we assume $\mathbb{L}3$ relationships, but our technique can adapted to deal with lower levels of precision as well.

*Capabilities and their constraints*

Figure 4 shows an excerpt from the full model of the example, with the *mass* capability, its constraint, alongside the related part of the FTG. *Matlab* is used as a formalism for defining the *mechanical model* of the system and has a capability of expressing *mass*. (That is, models being conform to the Matlab formalism, can have attributes of type *mass*.) Figure 4 shows how this aspect of the running example is modeled, with Equation 3 captured in a similar fashion as the other constraints were in Figure 3.

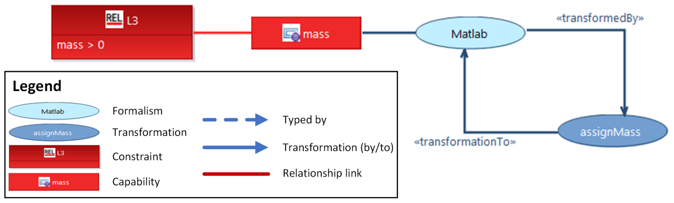


Figure 4: A capability and its constraint in the FTG.

The evaluation of such constraints, however, differs from the ones shown in Figure 3, as the constraints on *mass* are stemming from the universal laws of physics, while $m_T$, $m_P$, $m_B$, $m_M$ are specific to the system. To evaluate constraints of capabilities, we use the following rule.

*Definition 5 (Evaluation of capability constraints):* Any constraint applied on a capability imposes a constraint on every attribute typed by that capability.

This means, that based on the typing relationship between the *mass* capability and the system-specific masses $m_T$, $m_P$, $m_B$, $m_M$, Equation 2 can be unfolded as follows:

$$\begin{aligned} 0\ [kg] &< m_T \leq 150\ [kg], \\ 0\ [kg] &< m_P \leq 100\ [kg], \\ 0\ [kg] &< m_M \leq 50\ [kg], \\ 0\ [kg] &< m_B \leq 10\ [kg]. \end{aligned} \tag{4}$$

## E. Properties

As presented in [15] and [16], ontologies can be efficient enablers for inconsistency management in heterogeneous settings. During the translation of requirements to view-specific properties, each stakeholder keeps in mind certain domain properties, i.e. ontological properties. For example, an electrical engineer implicitly thinks about the capacity of the battery, a mechanical engineer reasons about how a battery would fit the frame of the AGV. Due to overlap in requirements, some ontological properties will be shared and/or will influence each other such that the related view-specific properties will be shared or influenced as well. Our framework allows defining property satisfaction relationships in terms of attributes. The satisfaction of a property can be inferred by checking the single constraints imposed on the specific attributes.

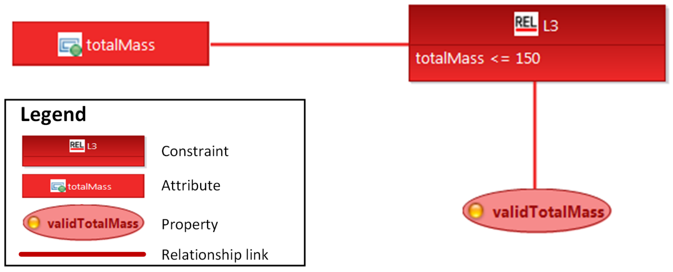


Figure 6: Excerpt from the example: the property *validMass*.

Figure 6 shows the property *validMass*. The satisfaction of the property is evaluated from the *totalMass* attribute, using the same constraint as the one shown in Figure 3 (i.e. the constraint on attribute $m_T$ in Equation 4), while the bottom right element, labeled with the name of the property, holds the boolean value of the satisfaction relationship.

This notion of properties allows various scenarios aiding inconsistency management, such as reusing domain knowledge from existing domain ontologies [17], using ontological reasoning [15] in conjunction with our techniques, and using contract-based design [18] [19] to aim co-design scenarios, i.e. parallel branches of the engineering process.

## F. Putting it all together: the process

Figure 5 shows the final model of the running example. In the middle, the yellow rectangles denote the activities of the PM with control flows in between them denoting the precedence relationship between the activities. On the right

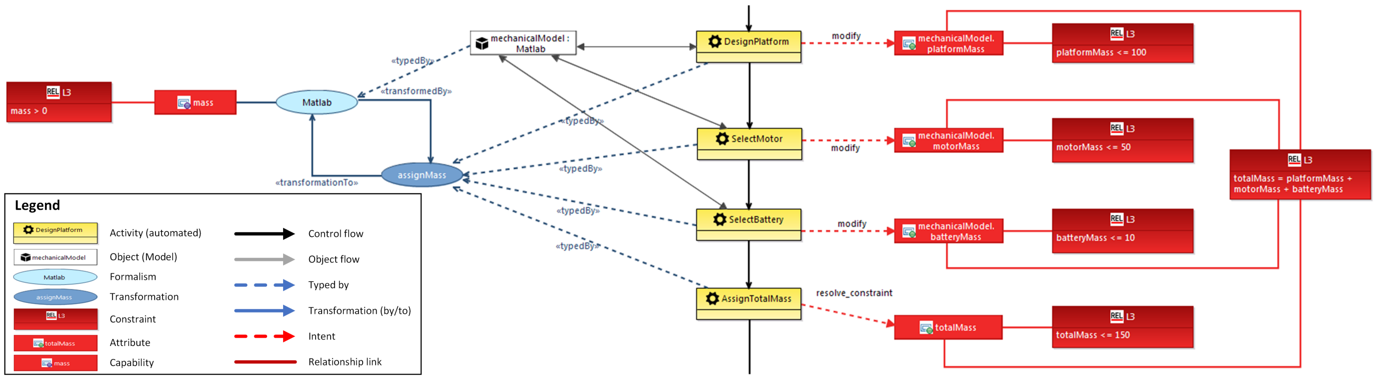


Figure 5: The FTG+PM based process model with the capabilities (left) and the attributes (right).

side, the attributes and constraints are shown. Activities and attributes are linked by *intents*, which express the purpose of the activity accessing a given attribute. The first three activities access attributes in order to *modify* them, while the last activity attempts to resolve a constraint wrt *totalMass*. Other types of intents include: reading the value of an attribute, imposing a constraint, locking/releasing an attribute in a parallel process branch, etc. This latter step is built into the process as an actual engineering step, but as shown in Section IV, in case of an inconsistency, the consistency monitoring service can stop the process before this point. In our previous work [7] we used *read-modify* pairs of intents to identify potential inconsistencies at the optimization phase. In this work, however, we leverage the notion of intents at run-time to narrow the scope of the consistency checking algorithm, i.e. to consider only the attributes which have been explicitly linked with an activity using a *modify* intent.

On the left side, the FTG and the only associated capability is shown. The typing relationships correspond to Figure 2:

- the mechanical model in the PM is an Object and it is typed by the Formalism *Matlab* in the FTG;
- the Activities are typed by the transformation *assignMass*;
- finally, the *mass* capability types all the masses on the right side, but this relationship is not visualized in the graphical view. (It is shown in a property view of the tool, however.)

Masses are assigned to the design when the respective activities are executed. Conceptually, this assignment can be viewed as a transformation of the model, and as such, the actual transformation logic is captured in the transformation typing the activities. In this case, *assignMass* holds the specification of the transformation. The activities operate on models. To check the consistency of the attributes, the attribute values are obtained by querying the appropriate models, i.e. the ones the specific attributes are persisted in. As Figure 5 shows, the name of the attribute is prefixed with the name of the model persisting the attribute. The first three attributes are all persisted in the *mechanicalModel*, which is a *Matlab* type of a model. Using this information, the querying is executed in the background by our tool via the Matlab API, without requiring the user to submit any extra information for this.

## IV. Managing Inconsistencies

In this section, we briefly present how the enactment of the previously modeled process (Section III) is carried out while enforcing a consistent state across the models. Our custom process enactment engine with fully modeled execution semantics is presented, as well the algorithm used for detecting inconsistencies during the enactment of the process. To facilitate the interplay in real engineering settings, we provide integration with multiple tools and frameworks, which will be discussed briefly as well.

### A. Architecture

Figure 7 shows the architecture of the process enactment engine. The engine is initialized by the *Process model*, defined previously in Section III. An explicit *Enactment model* augments the *Process model* with the notion of tokens and activity states (Figure 8), to be able to define the execution semantics. Execution semantics are defined by explicitly modeled *Transformation rules*.

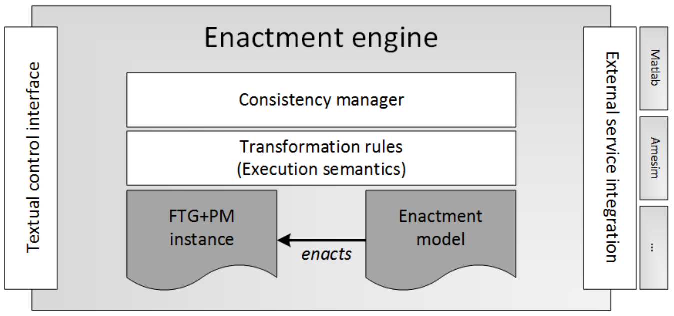


Figure 7: Architectural overview of the enactment engine.

The architecture has been implemented on top of the Eclipse platform. The Eclipse Modeling Framework (EMF) [9] is used for modeling purposes, while the model transformations have been realized using the VIATRA framework [11].

Activities of the process, especially the automated ones, often execute simulations and calculations over models on external storages by using external tools. For that, interoperability with a representative set of services is provided (*External service integration*). Our framework currently provides scripting support for Matlab/Simulink, and Amesim of Siemens/LMS through its native API. Executable pieces of Java code or Python scripts are supported and executed during the appropriate phases of the enactment.

A vital contribution of the stack is the *Consistency manager*, which features a symbolic solver for detecting inconsistencies. For this purpose, the SymPy [12] framework for symbolic mathematics is used. In Section IV-C, the algorithm of the solver is discussed in greater detail.

## *B. Execution semantics*

The execution semantics of the FTG+PM have been discussed previously in [20]. Here, we give a brief overview and focus on the main specificities in our current framework.

Since the core of enactment engine is fully modeled, the execution semantics are given by reactive live model transformations. Figure 8 shows the metamodel of the enactment engine. A *ProcessModel* (i.e. a full FTG+PM) is given to the compiler which creates an instance of the elements shown in green. During the enactment, a set of *Token*s define the marking of the process, i.e. the active *Node*s at a given moment.

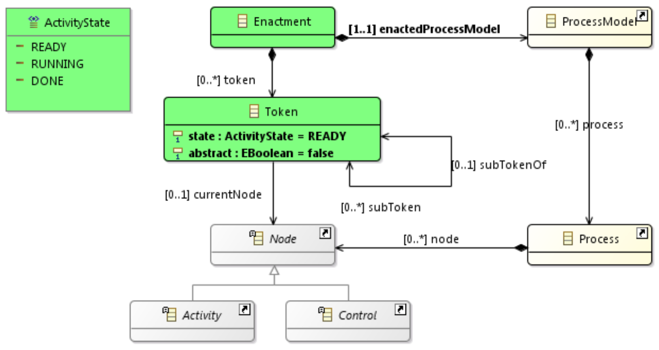


Figure 8: Metamodel for the enactment (green) along with the characteristic parts of the process metamodel.

A *Token* also equips *Activities* of the process with additional semantics regarding the state of their execution, modeled by *ActivityState*. This is required because the execution of *Activities* is not instantaneous.

- When a *Token* is moved to a new *Activity*, the *Activity* becomes *Ready*. The stakeholders and tools required to execute the *Activity* can be notified, the required models can be loaded into the tools.
- When the actual work in the *Activity* begins, the *Activity* becomes *Running*. This state can last for longer periods, especially in resource-intensive simulations or manual modeling activities, which may take days or weeks.
- When the actual work in the *Activity* is finished, the *Activity* becomes *Done* and the process can move on.

### *Transformation rules of the execution semantics*

*Definition 6 (Marking of the process):* By marking $M$ of the process we mean the function $M : N \to \mathbb{Z}$, where $N$ denotes the set of the *Node*s of the process with an integer number $\mathbb{Z}$ of *Token*s in it. A process is considered to be unmarked if there are no tokens present in it.

**Initialization** is a transformation which takes an unmarked process and transforms it into a process with an initial marking, i.e. with one token in its initial node.

**Finishing** is a transformation which takes a process with a final marking (i.e. every token in the final node) and transforms it into an unmarked process.

**Fork** is a transformation which takes exactly one token and produces a token for each parallel branch starting from that fork node. The input token is marked *abstract* (see Figure 8) and kept (hidden) in the model, while the newly created tokens are defined as *subtokens* of the input token, so that they can be identified once they have to be joined at the end of the parallel branches.

**Join** is a transformation which takes a token from each of its incoming parallel branches and joins those tokens. In a valid process model, the tokens to be joined must be the subtokens of the same (now abstract) parent token. The join is achieved by locating the parent token, placing it into the join node, marking it as not abstract, and removing the subtokens.

**Step** is a transformation which moves a token from a node to a consecutive node, while respecting the previous rules of forking and joining.

## *C. Algorithm for early inconsistency detection*

Early detection of inconsistencies requires computing the satisfiability of the system of constraints at certain points of the process. These computations are carried out on each *Step* in the process, based on Algorithm 1.

**Algorithm 1** Handling attribute modifications.

```
1: procedure STEP($token$, $nextActivity$)
2:     $token.currentActivity \leftarrow nextActivity$   ▷ Move the token to the next activity
3:     for all $i$:Intent, $a$:Attribute: $i(token.currentActivity$, modify, $a$, $v)$ do
4:         UPDATEATTRIBUTEVALUE($a$, $v$)   ▷ Assign value $v$ to attribute $a$
5:     end for
6: end procedure
```

On each *Step* in the process, the token is moved to the next activity (Line 2). As discussed in Section III-F, *intents* between activities and attributes help identifying the cases when an activity modifies the value of a property. For each of such intents (Lines 3-5), the attribute is updated and the change is propagated through the whole system of constraints. This latter step is being taken care of by Algorithm 2.

*Symbolic computation of constraints*

The updates to the system of constraints require introducing the new values of attributes and computing whether the constraints can be still satisfied later on in the process or not. Such a computation requires factoring in the potential impacts of the *future* attribute changes (explicitly modeled in the process). To execute these computations, we opted for the techniques of *symbolic computation*. Our main concern is the maintenance of a system of constraints by gradually simplifying them as attributes get updated, to the point, where contradictions appear in the equations, i.e. the set of potential solutions is empty, thus denoting an inconsistency in the system design. Alternative approaches include simulation of the process and abstract interpretation.

Algorithm 2 shows the steps taken in our symbolic computation approach. The algorithm is invoked by Algorithm 1 with the name and the new value of the attribute to be updated passed along as parameters. In *Phase 1* of the algorithm, the attribute-value assignment is translated to an equality constraint and added to the system of constraints (Line 2). In *Phase 2*, the algorithm propagates this change and attempts to simplify every constraint. This is achieved by iterating through the system of constraints (Lines 3-4) and factoring equation constraints (Line 5-6) into the rest of the constraints by trying to solve (simplify) the constraint (Line 7).

We use the SymPy [12] symbolic mathematics library to solve/simplify the constraints, thus the semantics is provided by the library. Constraints imposed by capabilities are calculated based on Definition 5 and applied on attributes.

Finally, in case an empty set is produced as a set of potential solutions for a constraint, we interpret it as an inconsistency and notify the user about this fact.

**Algorithm 2** Maintenance of the system of constraints.

```
 1: procedure UpdateAttributeValue($attribute$, $value$)
      Phase 1 – Impose a new constraint with equality
 2:     $model.constraints \leftarrow Eq(attribute, value)$
      Phase 2 – Propagation and simplification: substitute equality constraints into the rest of the constraints
 3:     for all $constraint1$ in $model.constraints$ do
 4:         for all $constraint2$ in $model.constraints$ do
 5:             if $constraint2$ is $Eq$ then
 6:                 $constraint1 \leftarrow constraint2$
 7:                 $solution = solve(constraint1)$    ▷ Try to solve the constraint
 8:                 if $solution = \emptyset$ then
 9:                     notify inconsistency
10:                 end if
11:             end if
12:         end for
13:     end for
14: end procedure
```

When an inconsistency is detected, the process is halted and cannot proceed until the inconsistency is not fixed. Resolving inconsistencies is outside the scope of the paper. A simple undo/redo functionality is provided by the framework, but more detailed research have been carried out by other authors, briefly discussed in Section V.

*Execution of the example*

We follow the process in Figure 5. During the execution, Equations 1 and 2 are maintained: whenever a value is assigned to an attribute present in one of the equations, the equations are simplified with that attribute. This means

- substituting the newly assigned value of the attribute to every occurrence of the attribute in every equation; and
- removing constraints without free attributes.

**Step 1: The platform mass is set to 100 kgs.** Activity *DesignPlatform* is executed and the mass of the platform is set in the mechanical model. Since the attribute is persisted in a Matlab model, the model is queried via the Matlab API for the value of the *platformMass* variable. The consistency manager uses this information to update the constraints with. The related constraint of Equation 2 ($0\ [kg] < m_P \leq 100\ [kg]$) is satisfied, and therefore the system can be simplified with it:

$$m_T = 100 + m_M + m_B[kg]$$

$$0\ [kg] < m_T \leq 150\ [kg]$$
$$0\ [kg] < m_M \leq 50\ [kg]$$
$$0\ [kg] < m_B \leq 10\ [kg]$$

Solving the constraints for $m_T$ results in a non-empty set of solutions:

$$100\ [kg] < m_T \leq 150\ [kg]$$

No inconsistency is detected, the process proceeds.

**Step 2: The motor mass is set to 50 kgs.** The *SelectMotor* activity is executed which sets the mass of the motor to 50 kgs.

$$m_T = 150 + m_B[kg]$$

$$0\ [kg] < m_T \leq 150\ [kg]$$
$$0\ [kg] < m_B \leq 10\ [kg]$$

At this point, Algorithm 2 detects the inconsistency sketched in Section II. Solving the constraints for $m_T$ results in an empty set of solutions:

$$150\ [kg] < m_T \leq 150\ [kg]$$

Since $0 < m_B$, it can be inferred, that after executing the next activity of the process, $m_T > 150$ will hold, which violates the constraint on the total mass. The process is halted and a notification is raised to the user to resolve the inconsistency.

## V. Related work

Model inconsistency is one of the main challenges in any engineering setting where more than one stakeholder is present. Di Ruscio et al [21] identify the research directions, challenges, and opportunities of collaborative MDSE and conclude, that inconsistency management is one of the main enablers of efficient collaboration. This challenge is exacerbated in scenarios of engineering systems of a *heterogeneous* nature, i.e. when the stakeholders come from different domains, work with different views on the system with their domain-specific formalisms and tools.

Multiple authors point out, managing inconsistencies should be carried out with processes in mind as well. Persson et al [22] argue that consistency between the various views of cyber-physical system design as one of the main challenges in design of such complex systems. This is due to relations between views, with respect to their semantic relations, process and operations which often overlap. Multi-paradigm modeling [5] advocates using the most appropriate formalisms, on the most appropriate level of abstraction, while also factoring in the processes manipulating the models. The framework presented in this paper, aims at the problem of inconsistencies with the processes in the focus.

In our work, we opted for the FTG+PM formalism for modeling processes. As compared to the widely used BPMN2.0 [23] or BPEL [24] based process modeling frameworks (e.g. jBPM [25]), our formalism allows modeling details more relevant to engineering scenarios in MDE settings. Models and transformations are first-class citizens in the FTG+PM, which enables deeper understanding of inconsistencies and more control over the enacted process.

Our methodology advocates making crucial attributes and constraints of the system explicitly modeled. Qamar et al [26] use a similar approach for inconsistency management by making model dependencies explicit. As opposed to our approach, the authors do not go as far as providing constraints for inconsistency management purposes, but use dependency links to notify stakeholders about *possible* inconsistencies when dependent properties/attributes change. It is a task of a stakeholder to verify the consistency of the models. In our approach, this is carried out in an automated way.

In the current paper, we do not focus on resolving inconsistencies of the models, but we provide undo/redo actions to revert to the latest consistent state. Eramo et al [27] present an approach where each of the consistent alternatives are maintained throughout the process and pruned when a decision is made and an alternative becomes infeasible. Such an approach can be viewed as a natural extension of our work, especially of the solver presented in Section IV-C. Mens et al [28] propose expressing the steps of inconsistency detection and resolution as graph transformation rules. Critical pair analysis is used to analyse potential dependencies between the detection and resolution of inconsistencies. The approach is efficiently handles cyclic inconsistencies, which is paramount in real system engineering scenarios and complements our work presented here. Almeida da Silva et al [29] investigate the possibilities of managing deviations of enacted processes from their respective specifications. It is not the scope of our work, but indeed, deviations from the specified process are big threat to the validity of any process-oriented engineering approach. The efforts put into analyzing and optimizing a process model can be easily demolished by deviating from (and sometimes even completely abandoning) the specification of the process. Egyed et al [30] investigate the impact of single inconsistency instances to the whole system by introducing the notion of change impact based scopes. Scopes are used to carry out resolution steps on the required regions of the models and thus enhancing the efficiency of the inconsistency management framework. Our formalism also supports the implementation of such scoping mechanisms, with the added potential of enriching the definitions of scopes with semantic information.

To implement our solver, we opted for the Python-based library SymPy. We briefly considered and researched two other libraries as well. SymJava[2] is a Java-porting of SymPy, but with a limited set of capabilities, which would have prevented us from implementing the second algorithm shown in Section IV-C. exp4j[3] is a library for evaluating expressions and functions in the real domain. Its main limitation is the inability of solving partial equations and inferring the (in)consistency based on that.

## VI. Conclusions

In this paper, we presented an approach for early detection of inconsistencies in complex engineering processes. The approach fits into a bigger inconsistency management framework, partially presented in our previous works [7][4][15].

The approach relies on modeling the characteristics of the system being developed, and using this information during the engineering phase to detect inconsistencies across the various engineering models of the system. The approach is process-oriented in a sense that attributes and capabilities of the system's models are modeled in conjunction with the actual engineering process, which then gets enacted. The enacted process is augmented with smart consistency checking algorithms, enforcing the consistent state of the design.

As presented, factoring ontological knowledge into the requirements of the system may shed light to additional constraints viable for early detection of potential inconsistent states of the system design. The main advantage of our approach is the support for such scenarios by uniformly handling instance- and meta-level constraints. As highlighted in the example, the advantages of such an early detection approach are visible already in very simplistic cases as the one above. The gain realized in real engineering processes can obviously be much higher, when the execution of resource and cost demanding activities can be prevented by the early detection of inconsistencies.

The proposed the modeling formalism enables lifting information relevant to inconsistency management purposes regarding the given process. Explicit modeling of such information is an enabler of improving the quality and efficiency of the process once enacted. Although the thorough modeling requires significant efforts from the stakeholders, it is needed

[2]https://github.com/yuemingl/SymJava
[3]http://www.objecthunter.net/exp4j/

to be done only once, before the actual design of the system commences. Such a front-loaded approach can be typically expected from companies on CMMI levels 3 and above [31]. As a consequence, our approach suits best the domain of complex heterogeneous systems, where the costs of dealing with inconsistencies is often in a different order of magnitude than the costs of modeling and optimizing the process.

A limitation of the framework may be its scalability, both from the user experience point of view (i.e. how efficient is it to model larger processes), and from the tooling point of view (i.e. how efficient is it to execute the optimization and monitoring of the enacted process). These concerns will be addressed in future work.

As a future work, we plan to combine the approach presented in this paper with our previous work [7]. This will enable explicit reasoning about the trade-off between managing inconsistencies in the process optimization phase and during the enactment. Another direction in our research is to support our approach with inconsistency resolution techniques. We aim for developing a semi-automated selection of resolution methods, which will require detailed cost models of the process and all of its aspects. Finally, the current framework serves as an enabler for our future research on inconsistency tolerance [4].

## Acknowledgements

This work has been partially carried out within the framework of the Flanders Make project *MBSE4Mechatronics* (grant nr. 130013) of the agency for Innovation by Science and Technology in Flanders (IWT-Vlaanderen). This work was partly funded with a PhD fellowship grant from the Research Foundation - Flanders (FWO).

The authors would like to thank the valuable insights and advices of Joachim Denil and Vera Farkas.

## References

[1] International Organization for Standardization, "ISO/IEC/IEEE 42010:2011, Systems and software engineering – Architecture description." https://www.iso.org/standard/50508.html. Acc.: 2017-07-07.

[2] A. Finkelstein, "A foolish consistency: Technical challenges in consistency management," in *Database and Expert Systems Applications*, vol. 1873 of *LNCS*, pp. 1–5, Springer, 2000.

[3] R. Van Der Straeten, "Inconsistency management in model-driven engineering," *An Approach Using Description Logics (Ph. D. thesis), Vrije Universiteit Brussel, Brussels, Belgium*, 2005.

[4] I. Dávid, E. Syriani, C. Verbrugge, D. Buchs, D. Blouin, A. Cicchetti, and K. Vanherpen, "Towards Inconsistency Tolerance by Quantification of Semantic Inconsistencies," in *COMMitMDE@ MoDELS*, pp. 35–44, 2016.

[5] H. Vangheluwe, J. De Lara, and P. J. Mosterman, "An introduction to multi-paradigm modelling and simulation," in *Proc. of the AIS'2002 conf. (AI, Simulation and Planning in High Autonomy Systems), Lisboa, Portugal*, pp. 9–20, 2002.

[6] J. Corley, E. Syriani, H. Ergin, and S. Van Mierlo, "Cloud-based Multi-View Modeling Environments," in *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, IGI Global, 2015.

[7] I. Dávid, J. Denil, K. Gadeyne, and H. Vangheluwe, "Engineering Process Transformation to Manage (In)consistency," in *Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016)*, pp. 7–16, http://ceur-ws.org/Vol-1717/, 2016.

[8] L. Lúcio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss, "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains," in *SDL 2013: Model-Driven Dependability Engineering*, vol. 7916 of *LNCS*, pp. 182–202, Springer, 2013.

[9] Eclipse Foundation, "Eclipse Modeling Framework (EMF) Website." https://eclipse.org/modeling/emf/. Acc: 2017-08-17.

[10] Eclipse Foundation, "Sirius Website." https://eclipse.org/sirius/. Acc: 2017-07-07.

[11] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró, "VIATRA 3: A Reactive Model Transformation Platform," in *Theory and Practice of Model Transformations*, pp. 101–110, Springer, 2015.

[12] SymPy Development Team, "SymPy Website." http://www.sympy.org/. Acc: 2017-08-17.

[13] I. Dávid, J. Denil, and H. Vangheluwe, "Towards Inconsistency Management by Process-Oriented Dependency Modeling," in *Proc. of 9th Int. Workshop on Multi-Paradigm Modeling*, pp. 32–41, 2015.

[14] J. W. Forrester, *Principles of Systems*. Productivity Press, 1968.

[15] K. Vanherpen, J. Denil, I. Dávid, P. De Meulenaere, P. Mosterman, M. Törngren, A. Qamar, and H. Vangheluwe, "Ontological Reasoning for Consistency in the Design of Cyber-Physical Systems," in *CPSWeek workshop proceedings*, 2016.

[16] O. Kovalenko, E. Serral, M. Sabou, F. J. Ekaputra, D. Winkler, and S. Biffl, "Automating Cross-Disciplinary Defect Detection in Multi-Disciplinary Engineering Environments," in *Knowledge Engineering and Knowledge Management*, pp. 238–249, Springer, 2014.

[17] S. Feldmann, K. Kernschmidt, and B. Vogel-Heuser, "Combining a SysML-based Modeling Approach and Semantic Technologies for Analyzing Change Influences in Manufacturing Plant Models," *Procedia {CIRP}*, vol. 17, pp. 451 – 456, 2014.

[18] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *European Journal of Control*, vol. 18, no. 3, pp. 217 – 238, 2012.

[19] K. Vanherpen, J. Denil, P. De Meulenaere, and H. Vangheluwe, "Ontological Reasoning as an Enabler of Contract-Based Co-design," in *International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems*, pp. 101–115, Springer, Cham, 2016.

[20] Denil, Joachim, "Design, verification and deployment of software-intensive systems: a multi-paradigm modelling approach," *University Antwerp*, 2013.

[21] D. Di Ruscio, M. Franzago, H. Muccini, and I. Malavolta, "Envisioning the future of collaborative model-driven software engineering," in *Proceedings of the 39th International Conference on Software Engineering Companion*, pp. 219–221, IEEE Press, 2017.

[22] M. Persson, M. Törngren, A. Qamar, J. Westman, M. Biehl, S. Tripakis, H. Vangheluwe, and J. Denil, "A Characterization of Integrated Multi-View Modeling in the Context of Embedded and Cyber-Physical Systems," in *EMSOFT*, pp. 1–10, IEEE, 2013.

[23] Object Management Group (OMG), "BPMN 2.0 Specification." http://www.bpmn.org/. Acc: 2017-08-17.

[24] OASIS, "WS-BPEL 2.0 Specification." http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. Acc: 2017-08-17.

[25] RedHat, "jBPM Website." https://www.jbpm.org. Acc: 2017-08-17.

[26] A. Qamar, C. J. Paredis, J. Wikander, and C. During, "Dependency modeling and model management in mechatronic design," *Journal of Computing and Inf. Science in Engineering*, vol. 12, no. 4, p. 041009, 2012.

[27] R. Eramo, A. Pierantonio, and G. Rosa, "Approaching Collaborative Modeling as an Uncertainty Reduction Process," in *COMMitMDE@ MoDELS*, pp. 27–34, 2016.

[28] T. Mens, R. Van Der Straeten, and M. D'Hondt, "Detecting and resolving model inconsistencies using transformation dependency analysis," in *International Conference on Model Driven Engineering Languages and Systems*, pp. 200–214, Springer Berlin Heidelberg, 2006.

[29] M. A. Almeida da Silva, R. Bendraou, X. Blanc, and M.-P. Gervais, *Early Deviation Detection in Modeling Activities of MDE Processes*, pp. 303–317. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[30] A. Egyed, "Automatically Detecting and Tracking Inconsistencies in Software Design Models," *IEEE Trans. on Sw. Engineering*, vol. 37, no. 2, pp. 188–204, 2011.

[31] CMMI Product Team, "CMMI for Development, Version 1.3, Tech. Rep. CMU/SEI-2010-TR-033," 2010.