Engineering Process Transformation to Manage (In)consistency

István Dávid
University of Antwerp & Flanders' Make, Belgium
istvan.david@uantwerpen.be

Joachim Denil
University of Antwerp & Flanders' Make, Belgium
joachim.denil@uantwerpen.be

Klaas Gadeyne
Flanders' Make, Leuven, Belgium
klaas.gadeyne@flandersmake.be

Hans Vangheluwe
University of Antwerp & Flanders' Make, Belgium
McGill University, Montréal, Canada
hans.vangheluwe@uantwerpen.be

ABSTRACT

Inconsistencies pose a severe issue to overcome in collaborative modeling scenarios, especially in settings with different domains involved. This is due to the significantly different formalisms employed that have overlapping semantic domains. A pertinent example are today's mechatronic and Cyber-Physical Systems.

In this paper, we propose an approach for managing inconsistencies based on explicitly modeled linguistic and ontological properties. We argue that to fully understand the reasons of their occurrence and impact on the overall design, inconsistencies should be investigated in the context of the process they emerge in. For this purpose, we propose a language for modeling processes in conjunction with the properties of the engineered system. Characteristics of inconsistencies are identified in terms of process models and properties. A method for optimal selection of management techniques is provided. We demonstrate our ideas on a case study of a real mechatronic system.

Keywords

inconsistency management, model-based design, cyber-physical systems, design space exploration

1. INTRODUCTION

Collaborative modeling scenarios are vulnerable to model inconsistencies. This is a consequence of the multiple views on the same virtual product that give rise to outdated and incorrect data. The problem is exacerbated when different engineering domains are involved, as different engineering domains use significantly different formalisms and paradigms to model specific parts of the system.

Overlaps in the semantic domain of models have been identified as the primary reason of model inconsistencies by many authors [27, 21, 33]. For example, the safety property of a mechatronic product translates to different concepts in terms of its mechanical, electric and control simulation models, and consequently, these concepts will overlap through the property of safety.

A significant amount of research has been dedicated to solving semantic inconsistencies on the syntactic level (for example [2, 6]). These approaches, however, are prone to lose vital information during the approximation and abstraction steps taken while translating semantic properties to linguistic structures and parameters. We argue that reasoning over explicitly modeled semantic properties suits the problem of tackling inconsistencies better, as demonstrated by Herzig et al [20] and Vanherpen et al [32].

Finkelstein [16] hints that instead of just removing inconsistencies, one should *manage* them. This entails reasoning about the causes and sources of inconsistencies, their evolution, interaction and impact on the overall design. We argue that this can be best achieved by investigating inconsistencies in the context of (i) the *design process* of the virtual product, (ii) the *modeling languages* and transformations used in the process, and (iii) the ontological and linguistic *properties* of the virtual product that are manipulated during the design. Explicit modeling of these concerns, along with the interactions between them, especially between the design activities and properties, enables better understanding of how inconsistencies arise and ultimately, how they should be managed, i.e. prevented, or detected and subsequently resolved.

In this paper we present an approach for inconsistency management in the context of engineering processes. Potential sources of inconsistencies are identified by considering characteristics of the process model. Management of inconsistencies is achieved by selecting the appropriate techniques from a catalogue of management patterns and applying them on the *unmanaged* process to achieve a *managed* one. Typical patterns include re-ordering activities of a process, ensuring property checks around inconsistency-prone regions and using design contracts [30]. Since the same type of inconsistency may be managed via different management patterns, the selection of the most appropriate one should happen through quantified cost measures. This problem is translated to a constraint solving and optimization problem which finds the best process alternative while managing every potential source of inconsistencies. The concept is shown in Figure 1.

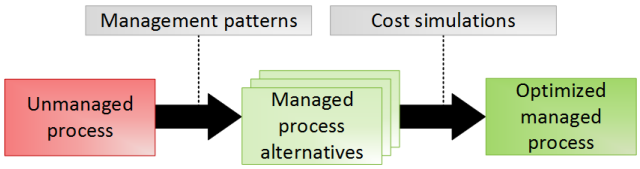


Figure 1: Conceptual overview of the approach.

There are two typical use cases of our approach: (i) *optimizing existing design processes* by first augmenting them with language and property information and then solving the constraint satisfaction and optimization problem; and (ii) *generating new processes* from language and property information. Both cases are realistic in complex engineering scenarios and their occurrence typically depends on the CMMI [9] level the engineering unit is situated on.

Contributions. The main contribution of this paper is an approach for managing inconsistencies in complex engineering processes, that includes

A a formalism (i.e. a language and its semantics) for modeling processes in relation with the properties of the engineered system;

B a method for detecting inconsistencies and identifying the most appropriate resolution technique based on quantified cost measures;

C a prototype tool supporting our approach; and

D a case study of a real mechatronic system.

The rest of the paper is organized as follows. In Section 2, we present a motivating case study of an automated guided vehicle. In Section 3 a formalism for modeling processes with properties is presented. In Section 4 we characterize inconsistencies in terms of the previously presented formalism, identify typical inconsistency patterns, provide an initial catalog for managing them and a method for selecting the most appropriate inconsistency management technique for single inconsistencies by multi-objective design space exploration (DSE). Section 5 discusses the prototype tooling supporting our approach. Finally, related approaches are reviewed in Section 7, and Section 8 concludes our paper.

2. CASE STUDY

We use a case study of the design of an automated guided vehicle (AGV) to motivate our work. The AGV is designed to transport payload on a specific trajectory between a set of locations. The drivetrain is fully electrical, using a battery for energy storage and two electric motors driving two wheels. Being a complex mechatronic system, the requirements of the AGV are specified by stakeholders of the different involved domains, such as (i) mechanical requirements: sufficient room on the vehicle to place payload; (ii) control requirements: following the defined trajectory with a given maximal tracking error; (iii) electrical requirements: 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 requirements: the previous requirements should be achieved at a minimal cost.

Figure 2 shows the conceptual geometric design of the AGV. The design team chose a circular platform, with two omniwheels in addition to the two driven wheels.
The design process needs to determine the sizing of the different components (motors, battery, platform) and tune the controller. This process is decomposed into multiple dependent design steps, such as motor selection, battery selection, platform-, controller-, and drivetrain design. The process requires an interplay between different domain-specific engineering tools, such as CAD tools for platform design,

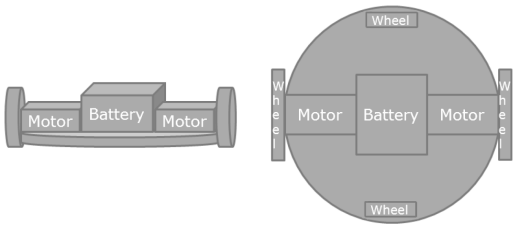


Figure 2: Front and top view of the conceptual design.

Simulink and Virtual.Lab Motion for multi-body simulations employed during controller design, AMESim for multi-physical simulations during drivetrain desing. 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 to overcome. By explicitly modeling linguistic and ontological properties and associating them with the engineering activities, patterns of inconsistencies can be identified and handled.

Running example

As a running example, we use a segment of the process in the case study shown in Figure 3. The example highlights how inconsistencies can occur due to properties of the system that interact with activities of an engineering process.

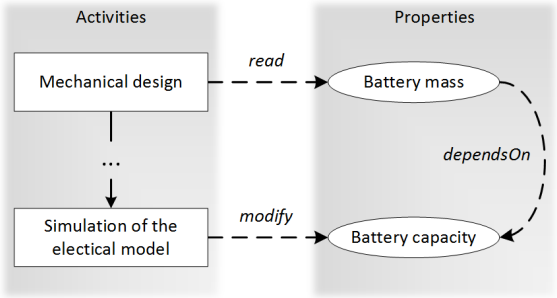


Figure 3: Running example.

Initially, components of the system, such as the battery, are selected based on approximations and domain expertise. The *mass* of the initially selected battery is considered during the *Mechanical design* phase to identify the mass constraints on other parts of the system. After the mechanical design phase, the electrical model is designed in details. This includes identifying the required *capacity of the battery* by *Simulating the electrical model*, in order to fulfill the autonomy requirement.

Inconsistencies may arise when the *Battery capacity* property is changed, because the *Battery mass* property *depends on* it: batteries with bigger capacity are typically heavier. As the capacity is changed, the mass becomes inconsistent with the capacity. Should the inconsistency get unnoticed, the engineered system will fail to meet the requirements.

There are two important specificities to this example that motivate our work.

First, inconsistencies occur due to the lack of explicitly modeled information about *how* activities access system properties. The key in identifying the above inconsistency is the explicit modeling of the nature of interaction between activ-

ities and properties, such as *reading* or *modifying* a property. We refer to this information as *intents* of activities over (a set of) properties.

Second, state-of-the-art techniques typically reason about inconsistencies in terms of *linguistic* model elements. The dependency between the two properties is, however, not persisted in any of the engineering models as it is an inter-domain relationship. To tackle this problem, we allow modeling *ontological* properties as well, and linking them to activities by intents.

3. A FORMALISM FOR MODELING PROCESSES WITH SYSTEM PROPERTIES

To model engineering processes with sufficient semantics for managing inconsistencies, we propose a formalism that augments the process with the syntactic and semantic *properties* that depict specificities of the engineered system.

We build our formalism on the FTG+PM [25] formalism, which enables the usage of process models ("PM") in conjunction with the model of languages and transformations (the formalism-transformation graph - "FTG") used throughout the process. As shown in Figure 4, languages and transformations serve as a type system to the processes: objects of the process are instances of languages of the FTG; and activities of the process realize transformations. Section 3.1 provides a brief overview on FTG+PM .

Figure 4: Relationships between processes, languages and properties.

We extend the above formalism by allowing explicit modeling of properties in context of processes and languages. We assume activities of an engineering process have a meaningful purpose of enhancing the system. This purpose is expressed as the *intent* of an activity with respect to a property or a set of properties. Furthermore, we extend the process model by enabling specification of *costs* of single activities. Modeling costs enables reasoning over the managed process candidates (as shown in Figure 1).

The type system plays an important role in the analysis of complex process models. Typing process objects by languages and linking activities to transformations enables reasoning about the MDE aspects of an engineering process with models and model transformations as first class artifacts. Additionally, in deployed and enacted process models, the language model serves as a basis for conformance and validity checks. For the sake of conciseness, we refer to the various elements of our formalism with slightly different terminology. In our terms, a process model PM consists of

- a set of processes P (equivalent to the PM part of an FTG+PM) that take
- a language model $LANG$ as a type model (equivalent to the FTG part of an FTG+PM), and relate to
- a property model $PROP$.

In the following, we elaborate on the specific parts of this process model. First, we briefly present the foundations of the FTG+PM formalism for typed processes; then we extend processes with costs; finally we discuss the property model in details.

3.1 Typed processes based on the FTG+PM

By the process $p \in P$, we mean a 4-tuple $(A, \Delta_c, O, \Delta_o)$ consisting of

- a set of activities (A);
- a set of directed control relations between activities $(\delta_c \in \Delta_c : A \mapsto A)$;
- a set of objects (O);
- a set of directed data flow relations between activities and objects $(\delta_o \in \Delta_o : A \mapsto O)$;

The control flow is a transitive relation, i.e. $\forall a_1, a_2, a_3 \in A : \delta_c(a_1, a_2) \wedge \delta_c(a_2, a_3) \Rightarrow \delta_c(a_1, a_3)$, and the following notation is used $a_3 \in \Delta_c^+(a_1)$.

The language model consists of modeling languages and transformations, formally: $LANG = (\mathcal{L}, \mathcal{T})$, and serves as a type system to processes, where languages type process objects and transformations serve as specifications to activities. In the case study, for example, the models of the mechanical design activity are typed by a *CAD language*, while the activity itself realizes the transformation(s) required to achieve the engineering goals during the mechanical design, such as dimensioning the platform of the AGV, and obtaining and executing a finite element simulation model. That is, $model_{Mech} \in O$, and $typeOf(model_{Mech}) = CAD$, where $CAD \in \mathcal{L}$; additionally, $typeOf(a_{MechDesign}) \in \mathcal{T}$.

3.2 Costs

We extend the definition of activities by their *costs*. For the sake of simplicity, we focus on costs constituting ratio scales, i.e. for the cost c of activity $a \in A$ the following holds:

$$c(a) : a \mapsto \mathbb{R}^+.$$

Costs serve as a basis for quantifying the differences between various process alternatives. In our current work, we approximate costs by the transition time required for single activities. Non-linear processes, i.e. the ones with directed loops, are typical in engineering scenarios. In these cases, the cost of a process is a non-deterministic value that can be obtained by appropriate simulations.

3.3 The property model

By a property model $PROP$ we mean a tuple (Π, R) consisting of

- the set of linguistic and semantic properties Π; and
- the set of influence relationships R between properties.

Properties of the running example in Figure 3 are the *Battery mass* and *Battery capacity*, while the dependency between those is a relationship with a *direction* and a *level of precision*. In the following, we first elaborate on properties (Section 3.3.1) and the influence relationships between them (Section 3.3.2); then we relate properties to processes by formalizing *intents* (Section 3.3.3); and finally, we provide a typing strategy for properties in terms of the FTG+PM formalism (Section 3.3.4).

3.3.1 Properties

Properties capture qualitative and quantitative characteristics of the modeled system. While linguistic properties represent *values* of various types, ontological properties capture system characteristics in terms of *satisfaction* constraints. The way properties are *checked*, also depends on their types. Checking a linguistic property means evaluating if the actual value of the property is within a range of acceptance criteria; checking an ontological property, on the other hand, means evaluating if the property is satisfied or not. While the former check is typically achieved by well-defined operators over the algebraic structures of the type of the property (e.g. arithmetic operators over number values), the latter type of checks typically involves simulations or model checking tasks.

For example, the battery mass property of the case study is a linguistic one (persisted in the CAD model), while safety and autonomy properties are semantic and can be checked by simulations or model checking techniques.

Explicitly modeling properties and their relationships enables (i) reasoning over these specificities, and (ii) fosters communication of tacit knowledge, which is especially important in the early phases of a multidisciplinary design process [32]. In our approach, ontological and linguistic properties are treated uniformly.

3.3.2 Influence relationships

Relationships between two properties are present if a change in one property potentially influences the other. In the case study, properties *Battery capacity* and *Current drawn* could be considered two properties with a relationship in between (Figure 5): a change in the *Battery capacity* will have an impact to the *Current drawn* and vica versa.

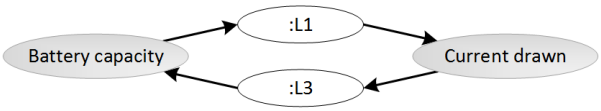


Figure 5: Influence relationship between properties.

A relationship r is formally defined as $r \in R = (\Pi_i^j \subseteq \Pi, \Pi_m^n \subseteq \Pi, \lambda)$, i.e.

- a set of influencer (input) properties $\Pi_i^j = \{\pi_i..\pi_j\}$,
- a set of influencee (output) properties $\Pi_m^n = \{\pi_m..\pi_n\}$,
- a level of precision $\lambda \in \{\mathbb{L}1, \mathbb{L}2, \mathbb{L}3\}$.

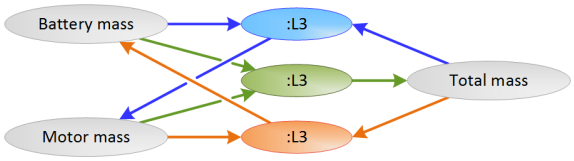
The three levels of precision are defined in our previous work [10] and are as follows.

- $\mathbb{L}1$: the **fact of influence** is known, its extent is not;
- $\mathbb{L}2$: **sensitivity information** between two values is known;
- $\mathbb{L}3$: the relationship can be expressed using an exact **mathematical relationship**.

Figure 5 shows a pair of properties that mutually influence each other, albeit on different levels of precision. A change in the *Current drawn* has an $\mathbb{L}3$ influence on the *Battery capacity* as follows: $BatteryCapacity \geq \int CurrentDrawn(t)dt$. The relationship in the other direction, however, cannot be determined in such details and thus, only constitutes an $\mathbb{L}1$ relationship. In the running example, the relationship between the *Battery mass* and *Battery capacity* constitutes an $\mathbb{L}2$ relationship: increasing the capacity requires increasing the battery mass, although the exact relation cannot be provided as batteries come in various architectures.

Acausal influence relationships

Acausality provides compactness in terms of the notation of relationships: it enables modeling of N-ary relationships in a more convenient and readable fashion. Figure 6a shows an N-ary influence relationship, depicting a simple law of physics: $BatteryMass + MotorMass = TotalMass$. By assigning value to two of the three properties, the third can be automatically calculated. The same information can be captured in an acuasal way as presented in Figure 6b.



(a) Causal notation of an N-ary relationship.

(b) Acausal notation of the same N-ary relationship.

Figure 6: Causal and acausal notation of a relationship.

In our approach, we allow using acausal relationships, but translate them to the causal equivalent form when carrying out analyses. Formally, given an acausal influence relationship r of level λ over a set of properties Π', the causality assignment maps $r = (\emptyset, \Pi', \lambda)$ onto a set of relationships $R' = \langle(\Pi'', \Pi''', \lambda)\rangle$, such that $\Pi' = \Pi'' \bigcup \Pi'''$.

The causal equivalent can be unambiguously determined only in symmetric N-ary relationships, meaning any M number of the N properties determine the remaining $N-M$, e.g. the one in Figure 6. In this case the causal equivalent will consisist of $\binom{N}{M}$ causal relationships.

Change scope of properties

By a change scope of a property π we mean a subset of properties $\chi_\pi \subseteq \Pi$ potentially affected by a change in π. Given two properties π_1 and π_2 directly linked by a relationship $r(\Pi_i^j, \Pi_m^n, \lambda)$, the change set is defined as follows.

$$\pi_2 \in \chi(\pi_1) \Leftrightarrow (\pi_1 \in \Pi_i^j \wedge \pi_2 \in \Pi_m^n)$$

That is, property π_2 is in the change scope of property π_1 iff π_1 is an input property and π_2 is an output property of an influence relationship. In the running example, the *Battery mass* property is in the change scope of *Battery capacity*.

The change scope is a reflexive and transitive relation, i.e.

$$\forall \pi \in \Pi : \pi \in \chi(\pi),$$

$$\forall \pi_1, \pi_2, \pi_3 \in \Pi : \pi_2 \in \chi(\pi_1) \wedge \pi_3 \in \chi(\pi_2) \Rightarrow \pi_3 \in \chi(\pi_1),$$

respectively. We use the following notation for the transitive closure of the change scope: $\pi_3 \in \chi^+(\pi_1)$.

3.3.3 Intents

Intents capture the *motivation* of an activity with respect to a property or a relationship of properties, such as *reading* and *modifying* a property. An intent $i \in I$ is defined by the tuple (a, s, t_I), where

- $a \in A$ is an activity;
- $s \in \Pi \bigcup R$ is the subject of intent, i.e. a property or a relationship;
- $t_I \in T_I$ is the type of intent.

We define four elementary intents for our approach: $T_I = \{read, modify, check, contract\}$, the first two being the typically occurring intents in standard engineering activities, while the latter two are specific to activities related to inconsistency management.

As discussed in Section 2, the main rationale behind explicitly modeling intents is that they carry valuable information regarding inconsistencies in processes, which enables reasoning about the origin and the potential management of inconsistencies. The inconsistency in the running example is possible to detect because of the exactly modeled pair of the read-modify intents on properties that influence each other and activities that are control-dependent. In Section 4 we formally characterize inconsistencies in terms of processes, properties and intents.

3.3.4 Typing of the property model

Intents relate properties to processes. In order to handle property models in a type-safe manner, however, properties and relationships have to be related to the type system defined by the language model as well.

We handle elements of the property model as special *process objects* that activities interact with. That is, following the definition of processes in Section 3.1:

$$\forall p \in P \; \forall s \in \Pi \bigcup R : s \in O(p),$$
$$\forall p \in P \; \forall i \in I : i \in \Delta_o(p).$$

Consequently, both properties and relationships are typed by appropriate languages, e.g. *OWL* languages [4], for modeling properties, or *graphs*, *algebra* and *Forrester system dynamics* [17] for modeling relationships.
Intents are typed by an *intent language* T_I, i.e.

$$\forall \; T_I = \{i_1..i_n\} : T_I \in LANG.$$

This means the language of intents in our approach can be aligned with the application domain of the problem at hand. The intents used throughout this paper are rather general and capture only access-change information over system properties.

4. MANAGING INCONSISTENCIES

After presenting the process modeling formalism for reasoning over inconsistencies in Section 3, we give a formal characterization of unmanaged cases that may lead to inconsistencies. We associate inconsistencies with *changes* in properties that are shared among multiple activities. Reasoning over such cases is enabled by explicitly modeled *intents* of the activities over the properties. To manage inconsistencies, we *augment the original process* with appropriate management patterns.

4.1 Formal characterization of unmanaged inconsistencies

In the following, we identify cases when inconsistencies may occur. We formalize this information in terms of pairs of activities, the related properties and intents. Generally, inconsistencies are introduced when an activity *modifies* a property that is accessed by another activity. A more formal definition can be given by distinguishing between activities situated in a sequential order and in parallel branches of a process. By sequential and parallel activities we mean

$$\forall a_1, a_2 \in A : seq(a_1, a_2) \Leftrightarrow a_2 \in \Delta_c^+(a_1);$$
$$\forall a_1, a_2 \in A : par(a_1, a_2) \Leftrightarrow a_2 \notin \Delta_c^+(a_1) \wedge a_1 \notin \Delta_c^+(a_2),$$

respectively. That is, two activities are said to be sequential iff one activity is transitively reachable from the other via the control flow of the process. If no such relation exists (in any direction), the activities are said to be parallel.

4.1.1 Sequential case

Given a pair of activities $a_1, a_2 \in A : seq(a_1, a_2)$, property π is said to be exposed to a potential inconsistency due to insufficient inconsistency management in the following case.

$$\exists(\pi' \in \Pi, i_1, i_2 \in I) : i_1(a_1, \pi, t_1) \wedge i_2(a_2, \pi', t_2) \wedge$$
$$\pi \in \chi^+(\pi') \wedge t_1 = read \wedge t_2 = modify \Rightarrow \exists \; ic(\pi) \in IC,$$

where IC denotes the set of unmanaged inconsistencies.
That is, property π is exposed to a potential inconsistency if activity a_1 first accesses it with a $read$ intent and subsequently activity a_2 $modifies$ property π', while property π is in the change scope of π'. The relation does not hold the other way around, i.e. by first modifying and subsequently reading a property does not lead to inconsistencies.

As a consequence of the reflexivity of the change scope, the above definition applies on cases where the same property is being read and modified as well.

4.1.2 Parallel case

Given a pair of activities $a_1, a_2 \in A : par(a_1, a_2)$, property π is said to be exposed to a potential inconsistency in the following case.

$$\exists(\pi' \in \Pi, i_1, i_2 \in I) : i_1(a_1, \pi, t_1) \wedge i_2(a_2, \pi', t_2) \wedge$$
$$\pi \in \chi^+(\pi') \wedge t_2 = modify \Rightarrow \exists \; ic(\pi) \in IC.$$

The definition is different from the one in the sequential case in not being specific about the type of intent i_1. This is because of the inconclusive ordering of a_1 and a_2 due to their parallel relation. Since the two activities may access the related properties in any order, the cases of potential inconsistencies cannot be narrowed to a specific ordering of read-modify intent pairs. That is, inconsistencies may arise if any of the two activities $reads$ property π, while the other one $modifies$ π'.

4.2 Patterns of inconsistency management

We use four typical inconsistency management patterns in our approach. This catalogue of patterns is, however, extensible in the prototype tooling.

4.2.1 Reordering and sequencing

Reordering and sequencing aim to modify the control flow in order to avoid inconsistencies.

Given a sequential case, i.e. $a_1, a_2 \in A : seq(a_1, a_2) \Rightarrow \pi \in IC$, the reordering strategy would swap a_1 and a_2, i.e. $seq(a_1, a_2) \rightarrow seq(a_2, a_1)$, to utilize that the appropriate order of read-modify intents does not lead to inconsistencies, as shown in Section 4.1.1.

In parallel cases, i.e. $a_1, a_2 \in A : par(a_1, a_2) \Rightarrow \pi \in IC$, the sequencing strategy would try every possible order of the activities and eventually select the one that leads to the most optimal process, i.e. $par(a_1, a_2) \rightarrow seq(a_1, a_2) \vee seq(a_2, a_1)$.

Reordering and sequencing are easy-to-apply and inexpensive patterns as they do not require introducing additional management activities. Both patterns work well in simple cases; in more complex processes, however, both patterns tend to introduce other inconsistencies.

4.2.2 Property check

Property checking is used to ensure no inconsistencies are introduced on specific sections of the process. A special activity a_{check} is added to the process that accesses the unmanaged properties with a *check* intent. If the result of the check is satisfactory, the process continues with the subsequent activities; in the case of a failed check, however, the process would fall back to the latest point where the inconsistency is not yet present and facilitate a re-iteration loop.

The property check pattern is a typically expensive management pattern as it introduces directed loops in the design processes and therefore, makes processes inherently nondeterministic.

4.2.3 Contracts

In a contract-based approach [32], the stakeholders would agree on acceptance criteria of specific properties *before* executing specific design activities. A special activity $a_{contract}$ is added to the process to represent the contract negotiation phase. The activity accesses the unmanaged properties with a *contract* intent. The contract is respected during the activities, thus providing means to avoid inconsistencies.

4.2.4 Assumptions

A less rigorous approach to contracts is also possible by making an educated guess about the shared properties. In the parallel case: $a_1, a_2 \in A : par(a_1, a_2) \Rightarrow \pi \in IC$, one of the parallel activities makes assumptions about the properties that will be modified by the other activity. However, these assumptions need to be checked once the process rejoins both branches. The benefit of the pattern is that only one of the branches has to be re-executed if the assumption proves to be invalid, i.e. an inconsistency may occur.

4.3 Process optimization

Since multiple management patterns can be applied for the same type of inconsistency with varying benefits, we formalize the process rewriting problem as a constraint solving and optimization problem as follows.

$$\begin{aligned} \underset{p}{\text{minimize}} \quad & \text{cost(p)} \\ \text{subject to} \quad & |IC| = 0, \\ & v(p) = 1. \end{aligned}$$

where $v(p) : p \in P \mapsto \mathbb{B}$ is the indicator function of the validity (i.e. well-formedness) of a process p. We also demand a fully managed process ($|IC| = 0$). We solve the problem by model transformation based multi-objective design space exploration (DSE) as shown in Figure 7.

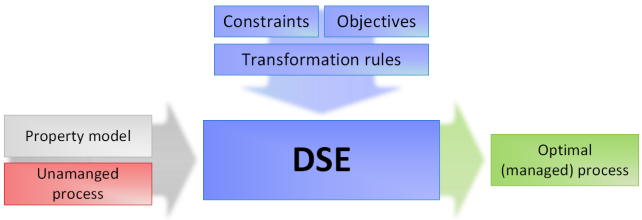


Figure 7: Detailed overview of the DSE approach.

The exploration mechanism takes the original *unmanaged process* and the *property model* as an input and produces an *optimal managed process* as a series of *model transformations* applied on the original process. (The property model is left intact as it reflects domain knowledge and as such, typically should not be changed because of a single process.) The exploration process is guided by mandatory *constraints* and optimality *objectives*.

4.3.1 Transformation rules

The purpose of using model transformations is twofold. We use them to augment the process with inconsistency management techniques, but also for rewriting the process into a better performing process. An example for the latter one is parallelizing as many activities as possible. Of course, this will affect the applicable inconsistency management techniques, and therefore, the execution and evaluation of these transformations must be achieved in a coupled way.

Management transformations

Transformation rules aiming to augment the process with inconsistency management techniques, are derived from the inconsistency patterns (Section 4.1) and management patterns (Section 4.2). A transformation rule is defined as

$$\begin{aligned} &\exists\, ic \in IC : ic(\omega \subseteq PM) \wedge \\ &\nexists\, m \in M : m(ic(\omega \subseteq PM)) \\ &\rightarrow apply(m'(ic(\omega \subseteq PM))),\ m' \in M. \end{aligned}$$

That is, if an inconsistency pattern ic is detected on a subset ω of the process model PM, and there is no corresponding management pattern m detected for the same subset of elements, then an appropriate management pattern m' is applied to the inconsistency.

General transformations

The overall goal of our approach is to find better processes, with respect to a goal function, that create correct products. Apart from the transformations specific to inconsistency management, therefore, we also use transformations that manipulate the structure of processes, such as adding and removing control flow between activities, arranging activities into sequences or parallel branches, etc. There is no restriction on how far the exploration strategy can go in restructuring the process, as it is determined in the exploration phase by considering that every inconsistency has to be managed. By making certain activities parallel, more linguistic and semantic overlap exists at the same instant in

the process and thus making the process more vulnerable to inconsistencies.

Note that certain applications of this pattern are not usable. For example, the parallelization of a design activity cannot be parallelized with the subsequent simulation of the created model.

4.3.2 Constraints and objectives

Constraints and objectives are used to guide the exploration process and evaluate the solution candidates. As defined previously, we constrain the set of solutions to processes that are valid and have no unmanaged inconsistencies.

As the objective function, the cost of the process is used. Since the cost of non-linear processes (i.e. the ones featuring directed cyclic graphs) is not deterministic, simulations of various kinds can be used to obtain the cost, such as event queueing networks or discrete event simulations.

5. PROTOTYPE TOOL SUPPORT

We support our approach with a prototype tool that allows (i) modeling processes with the aspects and semantics presented in Section 3; and (ii) augmenting processes with inconsistency management patterns, while identifying the optimal managed process.[1]

5.1 Process modeling

As a first step, the initial process has to be modeled along with the languages and properties. Our tool provides a graphical modeling environment for this purpose, implemented using the Sirius framework [13]. Figure 8 presents an excerpt from the process model of the case study, relevant for the running example discussed in Section 2.

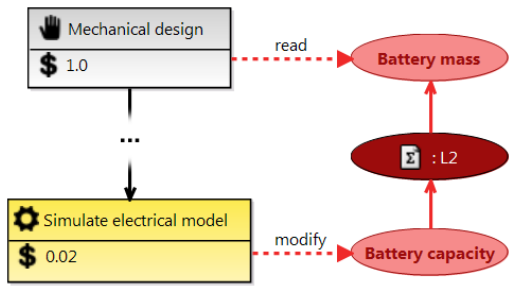


Figure 8: Excerpt from the process model of the case study.

The *Mechanical design* activity is modeled as a *manual* one, executed by the mechanical engineer. During the activity, the *Battery mass* property is accessed with a *read* intent. The cost of the activity reflects the approximate time required for executing it, i.e. 1.0 hour. Later in the process, the *Simulate electrical model* automated activity *modifies* the *Battery capacity* property which influences the *Battery mass*, as a consequence of the latter one being in the change scope of the former one. It is the explicit modeling of intents what enables identifying the potential inconsistency.

5.2 Management of inconsistencies

In the next phase the design space exploration is executed which includes

- matching the process model against the patterns of the *inconsistency catalogue*;
- applying patterns of the *inconsistency management catalogue* on the process; and
- selecting the best fitting management pattern based on *cost simulations*.

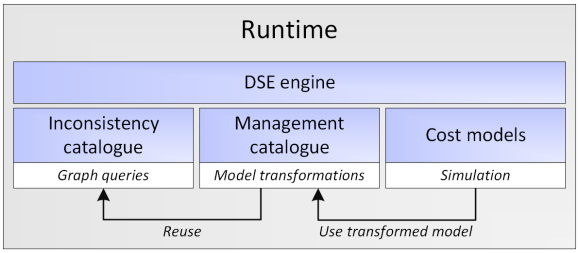


Figure 9: Conceptual overview of the core of the prototype.

For design space exploration, the VIATRA-DSE framework [1] is used. The inconsistency and management catalogues, as well as cost models are fully extensible, i.e. new inconsistency and management patterns can be formalized by the appropriate graph query and transformation languages, and costs can be evaluated using other approaches than the ones presented here.

5.2.1 Inconsistency catalogue

Patterns of inconsistencies are captured by graph queries over the input model. In our prototype tool, we use the VIATRA Query Language (formerly EMF-IncQuery) [31] for this purpose. Listing 1 presents the inconsistency pattern matched on the running example.

The pattern reflects the left-hand side (LHS) of the general transformation rule in Section 4.3.1. In Line 7, properties $p1 \in R^+(p2)$ and activities $a2 \in \Delta_c(a1)$ accessing the two properties with the respective *read* and *modify* intents are identified. Subsequently, in Lines 8-9, the number of applied inconsistency management patterns is determined. In case there is no management pattern applied (Line 10), the pattern is matched and the match set requires an inconsistency management pattern to be applied.

```
1 pattern unmanagedReadModify(
2   a1:Activity, p1:Property, a2:Activity, p2:Property)
3 {
4   find readModifySharedProperty(a1, p1, a2, p2);
5   checks == count find checkProperty(a2, _, p2);
6   contracts == count find contract(_, a1, p1);
7   check(checks+contracts==0);
8 }
```

Listing 1: The read-modify inconsistency pattern.

5.2.2 Management catalogue

Management patterns are captured as model transformations over the model. In accordance with Section 4.3.1, the LHS of the transformation rules consist of the previously defined inconsistency patterns; while the right-hand side (RHS) defines how the specific inconsistency should be handled by using one of the management patterns described in Section 4.2. Allowed LHS-RHS combinations are specified in terms of VIATRA Model Transformations [5], that enable directly reusing the previously defined graph queries as the LHS.

[1]The tool is available under the EPL licence at https://github.com/david-istvan/icm. The detailed case study is available at https://github.com/david-istvan/agv.

5.2.3 *Cost simulations*

As discussed in Section 3.2, cost of a process is not deterministic in most cases. We support our approach by two types of cost simulations in the prototype tooling. In *fixed iteration simulations*, the loops of the design process are identified and simulated with a fixed amount of iterations resulting in a process cost as follows

$$\forall p \in P : c(p) = \sum_{1}^{i=|A(p)|} c(a_i)n(a_i).$$

Loops are detected by a graph pattern matcher. If a loop is detected between two activities, the costs of activities in the loop are weighted by the number of iterations N and added to the sum cost. The parameter is to be set by a domain expert. In our experiments, we used 3–5 iterations as the typical values.

In event queueing network (EQN) based *stochastic simulations* [18], the decision of re-iterating over a loop is simulated with sampling from a probabilistic distribution. We carried out our early experiments using the SimEvents toolbox [26]. While stochastic simulations offer more precise results in terms of simulating the costs, they are also more demanding in terms of computation power and time.

5.3 Results of the case study

After exploring the space of alternative solutions and ranking them based on costs, the managed process is obtained. Figure 10 presents a managed alternative to the running example in Figure 8. As an allow-and-resolve type inconsistency technique, property checking is executed after the location of the potential inconsistency. The manual *CheckBatteryMass* activity accesses the potentially inconsistent property *Battery mass* with a *check* intent. Subsequently, a decision node is added to the control flow to enable a backward loop in case the check fails (*NO*) and to proceed if the check succeeds (*OK*). The pattern also introduces a loop that enables arbitrary re-iterations, which is the main contributor to the increased process cost. As opposed to this, an alternative reusing the technique of contract based design would not require such loops, but the management activity itself would be more elaborate as the contract negotiation phase requires stakeholders from multiple domains.

The real challenge of applying inconsistency management patterns in an orchestrated way, so that their application does not give rise to new unmanaged inconsistencies, is tackled by using a heuristic or exhaustive search through the state space. Applying our approach to the whole process of the case study resulted in a *fully managed process* with reasonable increase in costs. In our simulations, we measured up to 10% cost reduction while fully managing the process with two types of inconsistencies.

6. DISCUSSION

The results described in this paper serve as the foundations to a comprehensive inconsistency management framework. The approach, in its current form tackles two important problems in engineering practice. First, as presented in the previous sections, combining the notion of processes with inconsistencies sheds light on complex cases of inconsistencies that are otherwise not detectable nor manageable. Second, our approach enables expressing tacit domain knowledge explicitly and thus making it reusable across different

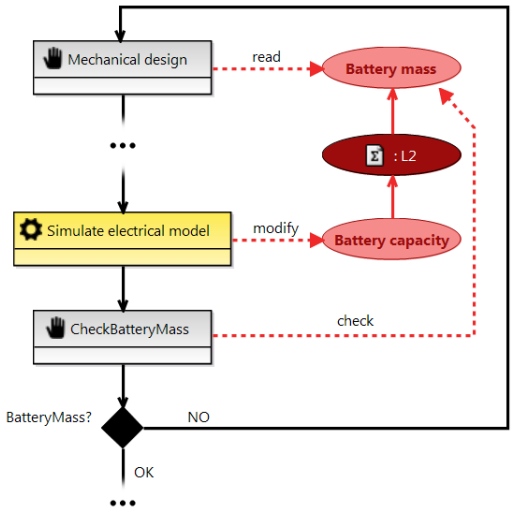


Figure 10: Managed alternative of the process in Figure 8.

processes (projects), at least partially, which is a typical concern in companies on CMMI levels 3 and above. [9] In order to enhance the reusing of domain knowledge, techniques of *ontological reasoning* will be investigated.

In the following, we discuss the required extensions to the current work in order to develop an inconsistency management framework.

6.1 Complex cost models and resources

As the primary research direction, we will extend the formalism in Section 3 with *more complex cost models* and the notion of *resources*. In our current approach, cost is a one-dimensional performance metric of the process, estimated from the development time. In real settings, however, different types of costs may be used to capture the performance of the process and thus providing a basis for optimization, e.g. queueing time, delay or processing volume [22]. By incorporating the notion of resources, the optimization problem will get extended by job scheduling aspects [3]. Resources pose additional constraints on process re-engineering in terms of the feasibility of the process.

6.2 Advanced solution techniques

The multi-objective nature of our DSE approach (Section 4.3) scales well with introducing additional dimensions to the underlying formalism discussed above. A known limitation of DSE based approaches is, however, the potentially missed global optimum of the input problem. We plan to address this limitation by translating the inconsistency management problem to more complex solution methods, such as heuristic algorithms [23] and genetic algorithms [34] that have been successfully applied in resource constrained process optimization.

6.3 Tooling aspects

The current version of the prototype tool supports the modeling phase of process engineering, but to achieve a comprehensive inconsistency management framework, the execution phase (i.e. the design phase of the virtual product) has to be supported as well. For this purpose, the process model will be *enacted* by using model transformations to provide the operational semantics. The explicitly modeled

formalisms/tools enable automated support for *tool interoperability*, with the potentially reusing integration frameworks such as OSLC [29].

7. RELATED WORK

Inconsistency management is a well-studied topic in the domains of software engineering, mechatronic design and cyber-physical systems, due to the typically multi-view and often multi-paradigm approach to system design. Persson et al [27] identify 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. Our technique embraces these ideas and addresses the problem of inconsistencies by explicitly modeling semantic properties and relating them to engineering processes.

Other approaches also acknowledge the role of semantic techniques in inconsistency management, and try to relate semantic concepts to the linguistic concepts of modeling. Hehenberger et al [19] organize structural design elements and their relations into a domain ontology to identify inconsistencies. A limited set of semantic properties are expressed with linguistic concepts which enables reasoning over semantic overlaps to a sufficient extent. Similarly, Chechik et al [8] introduce the notion of approximate properties: linguistic properties expressed as graph patterns which are accurate enough to appropriately approximate a semantic property. Approximate properties suitable to implement smart locking mechanisms in collaborative model-based design as they introduce a trade-off between the computational resources to obtain or check a property, and the accuracy of the results. As opposed to these, our approach makes semantic properties first-class artifacts and relates them to processes, instead of linguistic model elements, which enables management of a richer class of inconsistencies.

Ontologies have been used for inconsistency management by Kovalenko et al [24] to support automated detection of defects between domain-specific models. Similarly, Feldmann et al [15] use the OWL language in conjunction with a SysML-based approach to formally represent the design of a production system and evaluate the compatibility of domain-specific models in a collaborative setting. These approaches are complementary to ours: incorporating relationships between ontological properties for reasoning over inconsistencies is a planned extension to our work.

As opposed to the above techniques, inconsistency management in collaborative modeling is more frequently addressed on the linguistic level. Qamar et al [28] approach inconsistency management by making inter- and intra-model dependencies explicit. Dependencies are direct results of semantic overlaps and are used to notify stakeholders about possible inconsistencies when dependent properties change. Our approach introduces an indirection between models and properties by relating them to specific activities that during working over models also access properties with specific intents. Blanc et al [7] approach the detection of inconsistencies from a model operation based point of view, where models are stored as sequences of change events and inconsistencies are expressed in terms of CRUD operations. Our approach generalizes this approach by introducing intents that are analogous with model operations, but they express change operations in terms of activities and properties. Egyed et al [14] investigates the impact of single inconsistencies on 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. We carry out a similar scope detection and management on the property model of our approach. Specific technical challenges of collaborative modeling have been addressed by state-of-the-art techniques, such [11] for comparing and merging models and EMFStore [12] for model persistence. These techniques can serve as an implementational basis for improving our tool.

8. CONCLUSIONS

In this paper we presented an approach for managing inconsistencies in complex engineering settings from a process-oriented view. A modeling formalism has been provided to reason about how inconsistencies arise in processes and how they impact the overall design. For this purpose, the notion of *intents* has been defined as the explicitly modeled relationship between activities of processes and properties of the engineered system. We support the automated process of inconsistency management by a prototype tool built on Eclipse technologies. Finally, the approach has been evaluated over a case study of a real mechatronic system using our prototype tool.

Acknowledgments

The authors wish to thank *András Szabolcs Nagy* for his dedicated help with the VIATRA-DSE engine; *Tamás Szabó* for his critical and constructive comments; and *Kristof Berx* for his insights on the case study. 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).

9. REFERENCES

[1] H. Abdeen, D. Varró, H. Sahraoui, A. S. Nagy, C. Debreceni, Á. Hegedüs, and Á. Horváth. Multi-objective optimization in rule-based design space exploration. In *Proceedings of the 29th ACM/IEEE Int. Conf. on Automated software engineering*, pages 289–300. ACM, 2014.

[2] C. Adourian and H. Vangheluwe. Consistency Between Geometric and Dynamic Views of a Mechanical System. In *Proc. of the 2007 Summer Computer Simulation Conf.*, SCSC '07, pages 31:1–31:6, San Diego, 2007. Society for Computer Simulation Int.

[3] C. Artigues, S. Demassey, and E. Neron. *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*. ISTE, 2007.

[4] S. Bechhofer. OWL: Web ontology language. In *Encyclopedia of Database Systems*, pages 2008–2009. Springer, 2009.

[5] 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*, pages 101–110. Springer, 2015.

[6] A. Bhave, B. Krogh, D. Garlan, and B. Schmerl. Multi-domain modeling of cyber-physical systems using architectural views. *AVICPS 2010*, page 43.

[7] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th Int. Conf. on*, pages 511–520, May 2008.

[8] M. Chechik, F. Dalpiaz, C. Debreceni, J. Horkoff, I. Ráth, R. Salay, and D. Varró. Property-Based Methods for Collaborative Model Development. In *Proc. of 3rd Int. Workshop on The Globalization of Modeling Languages (GEMOC 2015)*, 2015.

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

[10] 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*, pages 32–41, 2015.

[11] C. Debreceni, I. Ráth, D. Varró, X. De Carlos, X. Mendialdua, and S. Trujillo. Automated Model Merge by Design Space Exploration. In *19th Int. Conf. on Fundamental Approaches to Software Engineering*, 2016.

[12] Eclipse Foundation. EMFStore Website. http://eclipse.org/emfstore/. Acc: 2016-07-19.

[13] Eclipse Foundation. Sirius Website. https://eclipse.org/sirius/. Accessed: 2016-07-19.

[14] A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *Software Engineering, IEEE Trans. on*, 37(2):188–204, 2011.

[15] 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}*, 17:451 – 456, 2014.

[16] A. Finkelstein. A foolish consistency: Technical challenges in consistency management. In *Database and Expert Systems Applications*, volume 1873 of *LNCS*, pages 1–5. Springer, 2000.

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

[18] D. Gross, J. Shortle, J. Thompson, and C. Harris. *Fundamentals of Queueing Theory*. Wiley, 2011.

[19] P. Hehenberger, A. Egyed, and K. Zeman. Consistency Checking of Mechatronic Design Models. In *30th Computers and Information in Engineering Conf.*, volume 3, pages 1141–1148. ASME, 2010.

[20] S. J. Herzig and C. J. Paredis. Bayesian Reasoning Over Models. In *11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014*, pages 69–78, 2014.

[21] S. J. Herzig, A. Qamar, and C. J. Paredis. An approach to identifying inconsistencies in model-based systems engineering. *Procedia Computer Science*, 28:354–362, 2014.

[22] Improvement Skills Consulting Ltd. Measuring Process Performance. https://ianjseath.files.wordpress.com/2009/04/measuring-processes.pdf, 2008. Acc.: 2016-07-19.

[23] R. Kolisch and S. Hartmann. *Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*, pages 147–178. Springer US, Boston, MA, 1999.

[24] 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*, pages 238–249. Springer, 2014.

[25] 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*, volume 7916 of *LNCS*, pages 182–202. Springer, 2013.

[26] MathWorks Inc. SimEvents Website. mathworks.com/products/simevents. Acc.: 2016-07-19.

[27] 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*, pages 1–10. IEEE, 2013.

[28] 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*, 12(4):041009, 2012.

[29] M. Saadatmand and A. Bucaioni. OSLC Tool Integration and Systems Engineering – The Relationship between the Two Worlds. *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 93–101, Aug. 2014.

[30] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. *European Journal of Control*, 18(3):217 – 238, 2012.

[31] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98, Part 1:80 – 99, 2015.

[32] 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.

[33] R. Wagner, H. Giese, and U. Nickel. A plug-in for flexible and incremental consistency management. *Proc. of the Int. Conf. on the UML*, page 93, 2003.

[34] M. B. Wall. *A Genetic Algorithm for Resource-constrained Scheduling*. PhD thesis, Cambridge, MA, USA, 1996. AAI0598050.