Verification and analysis of domain-specific models of physical characteristics in embedded control software

Arjan de Roo \textsuperscript{a,}, Hasan Sözer \textsuperscript{b}, Mehmet Akşit \textsuperscript{a}

\textsuperscript{a} University of Twente, Enschede, The Netherlands
\textsuperscript{b} School of Engineering, Department of Computer Science, Ozyegin University, Nisantepe Mevki, Orman Sk. No: 13, Alemdag/Cekmekoy 34794, Istanbul, Turkey

\textbf{Abstract}

\textbf{Context:} A considerable portion of the software systems today are adopted in the embedded control domain. Embedded control software deals with controlling a physical system, and as such models of physical characteristics become part of the embedded control software.

\textbf{Objective:} Due to the evolution of system properties and increasing complexity, faults can be left undetected in these models of physical characteristics. Therefore, their accuracy must be verified at runtime.

\textbf{Traditional runtime verification techniques that are based on states/events in software execution are inadequate in this case. The behavior suggested by models of physical characteristics cannot be mapped to behavioral properties of software. Moreover, implementation in a general-purpose programming language makes these models hard to locate and verify. Therefore, this paper proposes a novel approach to perform runtime verification of models of physical characteristics in embedded control software.}

\textbf{Method:} The development of an approach for runtime verification of models of physical characteristics and the application of the approach to two industrial case studies from the printing systems domain.

\textbf{Results:} This paper presents a novel approach to specify models of physical characteristics using a domain-specific language, to define monitors that detect inconsistencies by exploiting redundancy in these models, and to realize these monitors using an aspect-oriented approach. We complement runtime verification with static analysis to verify the composition of domain-specific models with the control software written in a general-purpose language.

\textbf{Conclusions:} The presented approach enables runtime verification of implemented models of physical characteristics to detect inconsistencies in these models, as well as broken hardware components and wear and tear of hardware in the physical system. The application of declarative aspect-oriented techniques to realize runtime verification monitors increases modularity and provides the ability to statically verify this realization. The complementary static and runtime verification techniques increase the reliability of embedded control software.

© 2012 Elsevier B.V. All rights reserved.

\section{1. Introduction}

Today, many embedded systems like digital document printing systems are largely controlled by software. The size and complexity of embedded control software is continuously increasing due to the demand for new functionality, more refined/optimized control, and increasing variety in the types of hardware that need to be controlled. Despite this trend, the control software must be kept reliable, even in varying circumstances such as changing context/environment, changing usage profile and evolution of system properties.

Embedded control software usually incorporates models that describe continuous physical characteristics of the system. We refer to these models as physical models in the rest of the paper. Apart from the system hardware model/architecture (process structure, memory organization, I/O units, the adopted cache/buffers, etc.), physical models comprise the physical properties and/or parameters of the controlled hardware components (e.g., heat, speed, power) and relationships among them (e.g., heat exchange). Physical models are needed in software control systems in order to represent the physical system with its actual parameters, to evaluate if the physical system deviates from the desired settings and to steer the physical system to eliminate the deviations if any. For example, several physical models have been implemented in the control software of digital printing systems to estimate the heat exchange among components like the toner belt and the paper path. The accuracy of such estimations is crucial for correct and optimal control.
of the system. However, these estimations are subject to faults that can trigger errors, which in turn can propagate and lead to a system failure [1]. The source of these faults can be different: (i) data received from (faulty) sensors can be inaccurate, (ii) there can be undiscovered faults in the implementation/model, and (iii) some models can become obsolete and invalid when the working context or system properties change (e.g., change, wear and tear of the hardware).

The correctness and accuracy of physical models depend on the physical circumstances in which the system is used. At design time, the system can be extensively tested under a wide variety of circumstances to ensure reliability. However, testing the system in all possible circumstances is impractical, if not impossible. Therefore, it is necessary to detect and cope with residual/transient defects at runtime. As such, runtime verification is necessary to check the accuracy of physical models for increasing the reliability of embedded control software.

Observers and monitors are the essential elements for runtime verification [2]. Traditionally, these elements are extracted/generated/defined based on specifications of events/states of interest in software and properties on these events/states. However, this approach is inadequate for verifying the physical models implemented in software. The monitors, in this case, are concerned with how models of physical characteristics are represented in software and whether these models of physical characteristics are consistent with physical reality. There are no formal properties of specification against which the implemented models of physical characteristics are verified. As such, it is not straightforward to specify monitors in terms of these concepts. Moreover, computations regarding physical models are typically implemented in a general-purpose programming language (GPL) and scattered across the implementation of several software modules, controlling various hardware. Therefore, it is hard to locate parts of the source code that are relevant for monitoring.

Previously, we introduced an approach for runtime verification of physical models in embedded control software [3]. In this approach, physical models are specified with a domain-specific modeling language (DSML), namely the SIDOPS+ language used in the 20-Sim toolset [4,5]. These specifications are composed with the control software implemented in a GPL (i.e., the C programming language) using the aspect-oriented Composition Filters model [6]. During execution, the correctness and accuracy of physical models are verified by exploiting the redundancy in these models (e.g., additional sensors, multiple relationships for the same physical variable), a technique inspired by the control engineering literature on state observers [7].

In this paper, we explain in detail how physical models are executed, composed with other software modules, and verified at runtime. We extend our approach to combine static and dynamic analysis. We statically analyze the composition between physical models and GPL modules by simulating the behavior of the specified composition filters. The results of such analysis can be used to detect conflicts in the composition, detect redundancy in the physical model and to optimize the execution efficiency of the composition filters. Our previous work [3] was focusing on error detection. In this work, we also introduce fault diagnosis to locate the root causes of detected errors. Finally, this paper describes the different application areas for the runtime verification approach, which cannot only be used for detected failures, but also for monitoring wear and tear, monitoring acceptable ranges of physical variables, detecting broken components in the system and calibration of the system.

To summarize, this paper provides the following contributions:

- The application of a DSML to explicitly specify physical models in software for the purpose of verifying their correctness at runtime and the application of the Composition Filters model to compose such physical models with other software modules specified in a GPL.
- Static analysis techniques to verify the composition of physical models in the DSML with other software modules specified in a GPL.
- A method to detect redundancy in physical models and utilize this to verify the consistency of physical models with physical reality.
- An approach to perform fault diagnosis when errors have been detected using runtime verification.

We have applied our approach on two industrial case studies for the analysis and runtime verification of digital document printing systems.

The remainder of this paper is organized as follows. In the following section, we provide background on formal verification and position our work with respect to the existing body of work. In Section 3 we introduce two industrial case studies. Section 4 motivates our research, illustrates the problems and challenges addressed by this paper in the context of the industrial cases. An overview of our approach is presented in Section 5. Section 6 introduces the DSML to specify physical models and the composition of DSML and GPL artifacts. Section 7 explains the static analysis applied on this composition for fault detection/diagnosis and optimization at design time. In Section 8, we discuss the runtime verification approach applied to detect errors. Section 9 explains the fault diagnosis technique that is applied upon the detection of an error. In Section 10, we evaluate our approach based on its application to the industrial case studies. We discuss several aspects of our approach in Section 11. Related previous studies are summarized in Section 12. In Section 13 we provide the conclusions and finally in Section 14, we explain the limitations of our approach and discuss possible future work directions.

2. Background: formal verification

There have been many formal methods and verification techniques devised and applied to ensure the quality and correctness of software systems [8,9]. The most popular techniques include static analysis [8], model checking [10] and theorem proving [9]. Although these techniques have been proven to be effective, it is not always feasible to apply them exhaustively and ensure a fault-free software. Even if the adopted software is fault-free, embedded systems are subject to the unreliability of commodity hardware and the adversity of operational environments [11]. Therefore, runtime verification is utilized as a complementary approach to further improve the reliability of software systems.

Traditional runtime verification approaches are based on dedicated languages to express the expected/correct behavior of a software system. These languages are used for specifying a sequence of events [12] like method calls [13] and they are based on different formalisms such as regular expressions [14] and linear temporal logic [15]. The specification is verified at runtime by monitoring the set of events based on a model of the system [2,16]. Examples of such runtime verification approaches are the MOP framework [17] and the tracematches extension to AspectJ [18].

Recently introduced runtime verification techniques are also concerned with the conformance of a concrete implementation with respect to a formal specification. There have been different formalisms and models used, such as regular expressions [19], abstract state machines [20], linear temporal logic [21] and UML models [22]. Regardless of the underlying formalism, the adopted formal specification mainly defines the correct order of events.

Our focus in this work is not the enforcement of correct order of events in a software system. Instead, we aim at verifying the
3. Industrial case studies

In this section we introduce two industrial case studies from the digital document printing systems domain. These case studies have been developed and evaluated together with our industrial partner (Oce) within the context of the Octopus project [23]. Although the selected case studies reflect only part of the overall system, they are still relevant for illustrating the problems and the effectiveness of our solution approach. They will be used as running examples of physical models that are in practice implemented as part of the embedded control software.

3.1. Case Study I: Warm Process

The first case involves a part of the printing process in digital document printing systems called the Warm Process. This process is responsible for transferring a toner image to paper.

3.1.1. System description

Fig. 1 gives a schematic overview of the parts in the printing system responsible for the Warm Process behavior. The Warm Process has two main parts; a paper path to transport sheets of paper and a toner belt to transport toner images. The contact point is the location where the paper path meets the toner belt. At this location the toner image is transferred from the toner belt to the sheet of paper. For correct printing, both the sheets of paper and the toner belt should have a certain temperature at the contact point. Therefore, the Warm Process contains two heating systems; a paper heater to heat the sheets of paper and a radiator to heat the toner belt. In addition, the physical system provides the following sensors and actuators:

- $T_{\text{contact}}$: Sensor that measures the temperature at the contact point.
- $v$: Actuator to set the printing speed.
- $T_{\text{ph}}$: Actuator to set the amount of power supplied to the paper heater.
- $P_{\text{rad}}$: Actuator to set the amount of power supplied to the radiator.

Note the variable $T_{\text{contact}}$ shown in the figure. This variable represents the temperature of the toner belt in the contact point. There is no sensor in the system to directly measure this value.

3.1.2. Adaptive control of the system

As part of the Warm Process, the speed is controlled, e.g., it can be decreased to reduce energy consumption or increased when printing on lighter paper (to increase productivity). If the speed of the system changes, the temperatures of the paper heater and belt also need to change, to maintain correct print quality. Engineers identified a relationship between the three variables: (i) speed ($v$), (ii) paper heater temperature ($T_{\text{ph}}$), and (iii) the temperature of the belt at the contact point ($T_{\text{contact}}$) that ensures correct print quality if the relationship holds. The relationship among these variables is given by the following equation (where $c_1$, $c_2$ and $c_3$ are constants):

$$ T_{\text{contact}} = c_1 \cdot v - c_2 \cdot T_{\text{ph}} + c_3 $$

The paper heater reacts slowly to changing temperature setpoints, while the radiator can quickly influence $T_{\text{contact}}$. Therefore, engineers...

---

1 Note that this only gives a simplified and abstract view of a printing system. In reality, there is no single actuator in the physical system to set the printing speed. Instead, the paper path consists of multiple motors and pinches, each of which can be controlled independently. If this control is done in a coordinated way, this leads to correct paper transportation at a certain speed. Note that a virtual speed actuator can be implemented in the control software. The control logic in the implementation of the virtual speed actuator can take care of controlling the different motors in the paper path to obtain the requested speed.
decided to mainly use the radiator to adjust the temperatures when the speed changes. This means that Eq. (1) is used to determine the required $T_{contact}$ (i.e., the setpoint).

As there is no sensor to directly measure $T_{contact}$, engineers had to identify the relationship between $T_{contact}$ and $T_{belt}$. They found the following equation (in which $c4$ is a constant):

$$T_{contact} = c4 \cdot \frac{P_{out}}{\sqrt{P}} + T_{belt}$$  \hspace{1cm} (2)

3.2. Case Study II: Drum Shuttling

The second industrial case is the Drum Shuttling subsystem of a printing system. The drum is a rotating cylindrical component in the printer system, on which the toner image is created. To reduce deterioration, the drum also has to shuttle (i.e., move backward and forward) along its axis.

3.2.1. System description

Fig. 2 schematically shows the drum and additional components needed to rotate and shuttle the drum.

There is a motor and gears for the rotational movement of the drum. We call this rotation $x$-movement, and the corresponding distance traveled by the surface of the cylinder the $x$-position.

The linear shuttling movement of the drum is provided by a stepper motor (i.e., a motor that rotates in fixed sized steps) and a cam, which is a component that can translate rotational movement into linear movement. We call this linear movement $z$-movement, and the corresponding position of the drum relative to the home position the $z$-position.

The physical system provides the following sensors and actuators:

- **pulseCount**: Sensor that counts and provides the number of hall pulses of the motor for $x$-movement. On each revolution, the motor gives a fixed number of hall pulses, so pulseCount is proportional to the number of revolutions made by the motor.
- **homeSensor**: Sensor that gives a signal when the drum is at a specific $z$-position (the home position).
- **dir**: Actuator to set the direction in which the stepper motor should step.
- **step**: Actuator that executes one step of the stepper motor when a signal is provided.

3.2.2. Adaptive control of the system

In the Drum Shuttling subsystem, the $z$-movement is performed relative with respect to the $x$-movement. This means that when the system is running at a lower speed, $z$-movement is also slower, meaning that less steps are performed per time unit. As no sheets can be printed when a step is performed, less steps per time unit is beneficial for the productivity. There is a function $f_{shuttling}$ to determine $zPos$ based on a given $xPos$: $zPos = f_{shuttling}(xPos)$. In principle, this is a linear function with the additional behavior that the slope is reversed at the boundaries of $zPos$ (to reverse the direction of movement).

For economical reasons there is no sensor to directly measure $xPos$. Instead, the pulses that the drum motor gives at a fixed rate per revolution are counted (pulseCount). The relationship between pulseCount and $xPos$ (rotation relationship) is described by the following equation:

$$xPos = \frac{\text{pulseCount}}{C_{pulsesPerRev}} \cdot C_{gearTransm} \cdot C_{drumCircumf}.$$  \hspace{1cm} (3)

Once the $xPos$ is obtained, it is transformed into a desired $zPos$ using the function $f_{shuttling}$. As there is no actuator to directly set the $zPos$, the desired $zPos$ needs to be translated into a desired stepPos of the stepper motor. The relationship between $zPos$ and stepPos (shuttling relationship) is given by the following equation:

$$\text{stepPos} = \frac{zPos}{C_{degreesPerStep} \cdot C_{shuttlingPerDegree}}.$$  \hspace{1cm} (4)

The Warm Process and the Drum Shuttling case studies are used in the following sections of the paper to illustrate the problems and challenges addressed in this paper and to evaluate the proposed approach.

4. Problem statement

The previous section showed that physical models have to be implemented as part of the embedded control software. These implementations, however, might not always be or remain accurate/correct. There are different reasons for this, e.g.:

1. The underlying assumptions on which the physical models are based might not be accurate enough. For example, the physical relationships might not accurately describe the physical reality.
2. The system might be used in different operational conditions than considered during design. For example, a printer system can be applied in different environmental conditions than expected, paper from a different manufacturer (having slightly different physical characteristics) can be used, etc.
3. Physical characteristics might change over time, because of, for instance, wear and tear of physical components in the system.
4. Physical characteristics might change because of evolution. For example, changes in the physical hardware influence the characteristics. If they are not updated accordingly in the implemented physical models, this can cause failures. If the physical models are implemented in a GPL and tangled with
core behavior of components, updating them can be error-
prone, because it is hard to locate the affected parts in the code.
5. The engineer implementing a physical model might introduce a 
fault.

Faults in an implemented physical model may lead to failures in 
the behavior of the system. In the embedded software domain, just 
as in many other domains, it is not possible to test the software in 
all possible physical conditions. Therefore, certain faults might re-
main undetected. Therefore, we need to verify physical models at 
runtime and there are two important challenges to realize this ver-
ification in embedded software, as explained in the following 
subsections.

4.1. Verification without formal properties

Faults in implemented physical models lead to observable fail-
ures in the physical behavior of the system, but not necessarily to 
observable failures in software behavior. This hinders the applica-
tion of common runtime verification techniques that focus on 
monitoring software behavior only.

As discussed in Section 2, runtime verification is commonly 
used to verify whether the software implementation satisfies a 
number of formally specified properties, or is consistent with a for-
mal specification. However, to verify the consistency of physical 
models with physical reality in embedded control software at run-
time, one does not verify these models against a formal specifi-
cation of physical reality, as there is no such formal specification. 
Therefore, we need a new approach to verify that physical models 
in embedded control software are consistent with physical reality.

4.2. Physical models implicit in program code

The physical models are often expressed in a GPL, such as C++, 
and tangled within control components. This is shown in Listing 1 
for the radiator controller of the Warm Process case study. This 
code fragment shows code that is executed for each iteration of 
the control loop. The control logic is shown on Lines 14–16. Lines 
10 and 11 show the implemented physical model.

This implicit implementation of physical models and tangling 
with core component code makes the models hard to identify, ana-
lyze and verify.

In the following, we provide an overview of our solution ap-
proach for this problem.

5. Approach overview

This section introduces an overview of our approach. The fol-
lowing sections explain the involved steps in more detail. Fig. 3 
schematically shows our approach. Hereby, the numbers on arrows 
indicate the order of activities. The set of activities, related tools 
and contributions can be grouped into three parts: (i) extension 
of composition filters to enable the separate specification of phys-
ical models, (ii) static analysis for verifying internal consistency 
and correct composition, and (iii) interpretation and verification 
of physical models at runtime. In the following, we explain each 
of these three parts of the approach.

5.1. Separately specifying physical models using a DSML

One of the problems for runtime verification of physical models 
mentioned in Section 4 is the fact that physical models are implic-
itly implemented in program code, and therefore hard to analyze 
and verify as physical models. To overcome this problem, our ap-
proach involves the separation of physical models from other 
application logic written in a GPL, by using a domain-specific mod-
eling language (DSML) to specify the physical models and the Com-
position Filters model to compose DSML models with other 
application logic written in a GPL.

Fig. 3 shows three different types of artifacts: software modules 
written in a GPL, physical models specified in a DSML and compo-
sition filters. The GPL software modules are written by software 
engineers. They are executed in a runtime environment for the 
GPL. In the following, we call a specification written in a general-
purpose programming language a (GPL) module or base program. 
Physical models are specified in a DSML by software engineers 
and/or domain experts. We adopt the DSML SIDOPS+ from the 
20-Sim toolset to define physical models. 20-Sim is a widely ap-
plied toolset with an extensive set of functions suitable for model-
ing and simulating physical systems[24,4]. In the following we call 
a SIDOPS+ specification a physical model, or just model.

To compose physical models with GPL modules, we apply the 
Composition Filters model by extending it with the execution 
semantics of physical models. Section 6 explains in more detail 
how physical models are specified using a DSML and composed 
with the base program using the Composition Filters model. 
Although this part is not the main focus of this paper, separating 
physical models from the control software and specifying them 
with a DSML enables us to perform domain-specific static and dy-
namic analysis.

```
double PI_rad_I = 0d;
/
/* Control loop */
// Get data using interfaces
double w = io.getV();
double Tph = io.getTph();
double Tbelt = io.getTbelt();

// Print Quality and Belt Temp. physical characteristics
TcontactSP = c1*w - c2*Tph + c3;
Tcontact = c4 * Prad / sqrt(v) + Tbelt;

// PI control logic
double deviation = TcontactSP - Tcontact;
PI_rad_I = PI_rad_I + deviation;
double PI_rad = c6 * deviation + c7 * PI_rad_I;

// Send value of Prad to Physical System I/O module
io.setPrad(Prad);
```
6. Separation and composition of physical models

In this section we introduce our approach to separate physical models from other application logic using a DSML and to compose the resulting DSML models with other application logic in GPL modules using the Composition Filters model. As this part of the approach is not the main focus of this paper, we introduce here only the basics that are necessary to follow the next sections. More detail about how physical models are separated, composed and executed at runtime can be found in [25].

6.1. Specifying physical models in a DSML

6.1.1. The 20-Sim/SIDOPS+ language

The 20-Sim toolset is used to model and simulate physical systems. Part of the 20-Sim toolset is the language SIDOPS+. With this language it is possible to mathematically define physical models. The SIDOPS+ language also offers a composition mechanism to compose smaller physical models into larger physical models. Besides the SIDOPS+ language, the 20-Sim toolset provides a modeling environment to model physical systems with iconic diagrams and bond graphs. For brevity these features are not discussed in this paper, as they result in equivalent representations [5].

Listing 2 shows an example specification in the SIDOPS+ language. This specification contains three types of definition blocks: constants, variables and equations. SIDOPS+ provides more language constructs to model physical processes, but because they are not used in this paper, they are not explained here.

Constants are defined in the constants definition block. The example shows the definition of the constant pulsesPerRev of type real. SIDOPS+ supports a number of different types, such as integer, real and boolean. Constant definitions always have a value assignment. In the example, the value 24.0 is given to pulsesPerRev.

The variables block defines the physical variables. The example shows the definition of two physical variables: pulseCount and motorRotation. Variables have a type. They can also have the modifier global, which indicates that the same variable can also be used in other models. In a 20-Sim simulation, this means that if this variable is defined in multiple submodels, composed into a larger model, they all represent the same variable.

The example shows that a variable may have an initial value assigned. This assignment is optional. Furthermore, it is also possible to attach the name and unit of the corresponding physical quantity to the variable, as the example shows for the variable motorRotation: within curly braces the quantity name Rotation and unit rev are given. The definition of the quantity name and unit is optional, but can be used to check whether defined equations are consistent. The quantity name and unit can be attached to constant definitions in the same way.

Equations are defined in the equations block. Equations specify mathematical relationships between variables. An equation is composed of two expressions, separated by an equality (=) sign. An expression can contain variables, constants, operators and predefined functions. The example shows how the variables motorRotation and pulseCount relate to each other.

### Listing 2. 20-Sim example specification.

```plaintext
1. constants
2.   real pulsesPerRev=24.0;
3. variables
4.   integer global pulseCount=0;
5.   real global motorRotation (Rotation, rev);
6. equations
7.   motorRotation=pulseCount / pulsesPerRev;
```
For further detail about the SIDOPS+ language, we refer to the 20-Sim documentation in [5].

6.1.2. Composing multiple physical models

Multiple models, resulting from different SIDOPS+ specifications, can be composed into larger models, expressing more complex physical systems. Composition is basically performed by including multiple SIDOPS+ specifications in a physical model. The physical variables with modifier `global` provide the interaction points between the submodels in the composition. This means that if multiple submodels in the model define a variable with the same typing and the same name and they have the modifier `global`, then this represents a single variable in the composed model. If a submodel defines a variable without the modifier `global`, then in the composed model this variable is different from any variable defined with the same name and typing in another submodel. The next example illustrates composition of physical models.

Listings 3–5 show three SIDOPS+ specifications for respectively the physical components `motor`, `gears` and `drum` from the Drum Shuttling case study. These SIDOPS+ specifications will be composed into a physical model of the interaction between the motor, gears and drum.

The listings show that the variable `motorRotation` provides the interaction point between the submodels `motor` and `gears`. Variable `gearRotation` provides the interaction point between the submodels `gears` and `drum`.

6.1.3. Physical model instances

A physical model defined in SIDOPS+ describes mathematical relationships between physical variables. But, a physical model does not represent the values of the physical variables at a certain moment in time. We call the valuation of the physical variables in a physical model at a certain moment in time a physical state. A physical model then defines a set of possible physical states, i.e., all physical states in which the relationships in the model are valid.

To use physical models in embedded control software, the physical state should be represented in software. Furthermore, the consistency of the physical state with the corresponding physical model should be maintained. This means that the mathematical relationships in the physical model are valid for the given values of the physical variables in the physical state. To represent the physical state and maintain its consistency with the physical model, we introduce the concept of a physical model instance. A physical model instance maintains a physical state for a given physical model. The physical state maintained by a physical model instance can be manipulated. The runtime ensures consistency of the physical state with the physical model. Fig. 4 shows schematically the relationship between the physical model and the physical model instance.

Depending on the context in which physical model instances are used, they model different states of the system, for example:

- **Current system state:** The most straightforward application of physical model instances is to model the current system state. Sensor readings are used to update the state in the physical model instance and to maintain its consistency with the real physical machine. For example, in the Drum Shuttling case study this can be applied to determine the current `xPosition` of the drum.

- **Desired system state:** A second application of physical model instances is to model a desired system state. For example, this physical model instance can be used by higher-level controllers to communicate a desired value of a certain physical variable. Using the mathematical relations in the physical model, the desired value of this physical variable is translated into desired values of other physical variables, which might act as setpoints.

```plaintext
1 constants
2   real pulsesPerRev=24.0;
3 variables
4   integer global pulseCount=0;
5   real global motorRotation (Rotation, rev);
6 equations
7   motorRotation=pulseCount / pulsesPerRev; // eq1

Listing 3. SIDOPS+ model of motor rotation (motor).

1 constants
2   real gearTransmission=15.0 / 126.0;
3 variables
4   real global motorRotation (Rotation, rev);
5   real global gearRotation (Rotation, rev);
6 equations
7   gearRotation=gearTransmission * motorRotation; // eq2

Listing 4. SIDOPS+ model of gear rotation (gears)

1 constants
2   real drumCircumference=350.0 (Distance, mm);
3 variables
4   real global gearRotation (Rotation, rev);
5   real global drumRotation (Rotation, rev);
6   real global xPosition (Distance, mm);
7 equations
8   drumRotation = gearRotation; // eq3
9   xPosition = drumRotation * drumCircumference; // eq4

Listing 5. SIDOPS+ model of drum rotation (drum).
```
for lower level controllers. For example, in the Drum Shuttling case study, this can be applied to determine the desired stepPosition of the stepper motor, based on the desired zPosition of the drum.

- A state representing control constraints: This application of physical model instances is a combination of the other two example applications. The physical state is a mixture of the current system state and desired system state, to determine setpoints for certain controllers. This can be used to enforce constraints on the desired state of the physical system. These constraints are basically physical models that determine desired values for certain physical variables based on the current values of other physical variables. For example, suppose a physical model contains the relationship \( a = 2 \times b \). Suppose that the value of \( b \) in the physical model instance is the current value in the system and is updated using a sensor. Suppose that the value of \( a \) in the physical model instance (which is obtained using the equation) is the setpoint for a controller that controls \( a \). Then the equation is a constraint on the system that ensures, using the sensor and the controller for \( a \), that \( a \) equals \( 2 \times b \). Such an application of physical model instances can be used in the Warm Process case study, to determine the required value \( T_{\text{contact}} \) (i.e., \( T_{\text{contact}}^\circ \)), based on the current values of \( T_{PH} \) and \( v \).

6.2. Composing physical models with GPL modules using the composition filters approach

In this subsection, we show how a physical model instance is composed with modules in the GPL, using the composition filters approach from the Compose\(^{1}\) language. For detailed information on the composition filters model, the language and its application in runtime verification in general, we refer to [6,26–28]. Fig. 5 shows how the composition filters approach is applied to physical model instances.

Artifacts on which composition filters are applied consist of two parts: an implementation object and an interface part. The implementation object contains the internal semantics of the artifact (e.g., the instance variables and implementation of methods if the artifact is an object in an object-oriented language). The interface part represents the interaction semantics of the artifact, i.e., the way in which the artifact can interact with other artifacts (e.g., the public interface of objects in an object-oriented language).

When composition filters are applied on physical model instances, the implementation object contains the physical relationships defined in the physical model and the physical state that is part of the physical model instance. The interface part consists of an event model and a base interface. The base interface provides a message-based interface to query and update physical state information from the physical model instance. The event model defines a number of events within the execution of a physical model instance that can be of interest to external artifacts.

A composition filter applies matching on events and defines certain behavior to be executed when an event matches. An example of such behavior is a dispatch of a message to a GPL software module. Certain composition filters return a value, which represents the new value of the physical variable to which the event applies.

6.2.1. The event model

The event model defines the types of events that can occur within the execution of a physical model instance and that can be matched within a composition filter. It is a generic definition applicable for all physical model instances, not specific for a particular physical model instance.

Table 1 shows several properties of events defined by the event model. Matching within a composition filter is performed using these properties.

6.2.2. Defining composition filters

Listing 6 shows an example composition filters specification. This specification is applied to the physical model instance of the Drum Shuttling case study that contains the drum rotation model specified in Listings 3–5.

In this case, the pulseCount is retrieved using a sensor. If the xPosition is changed, the new value is given to the ShuttlingController module, which implements the control logic to translate the xPosition of the drum to the desired zPosition of the drum.

Lines 1 till 10 show the definition of the filter module Rotation10. This filter module consists of two references to external objects: pcSensor is the external Sensor object to provide pulseCount, shControl is the external ShuttlingController object to which a new xPosition can be given.

The filter module also defines two composition filters. On Lines 6 and 7 the pcFilter has been defined. The type of this filter is Dispatch, which indicates that when the filter matches, a message is dispatched to a GPL module. The matching part of the filter shows that the filter matches for Request events concerning
pulseCount (the property `variableName` has value `pulseCount` and property `eventType` has value `Request`). After the matching part, this filter has an assignment part (shown on Line 7). In this assignment part, the target and selector of the dispatch message are provided. The target is the `pcSensor` external object; the selector is the method `getValue`. If the filter matches, a dispatch will be done to this method, and the resulting value will automatically be stored in the `returnValue` property.

Lines 6 and 7 show the definition of the second composition filter, which matches on `Update` events of `xPosition` and dispatches to the method `setXPos` in the `shControl` external object.

Lines 12 until 17 show a superimposition definition. In this superimposition definition, the defined filter module `RotationIO` is placed on the physical model instance of the physical model that contains one or more of the SIDOPS+ specifications `motor`, `gears` and `drum`.

7. Static composition analysis

In the previous section, we have introduced composition filters for composing physical models (specified in a DSM) with GPL software modules. This section describes the set of static analysis techniques that we apply on such composition filters.

7.1. Background: composition filter reasoning

A set of composition filters (filter set) that is superimposed on an object or physical model instance defines certain behavior on messages/events. The filter set can define different behavior for different messages, by matching and selecting based on the properties of the message/event. The behavior of a specific message/event in a given filter set can be statically determined. Static analysis of the behavior of messages/events in a filter set has been extensively investigated in [29] for the following purposes:

- To detect consistency conflicts in the filter set, for example unreachable filters and matching parts that never match.
- To analyze interface changes: if composition filters are superimposed on objects, they can change the interface to these objects.
- To compile the filter set to GPL code, for efficient execution.

The general approach in composition filter reasoning [29] involves the following steps:

1. Transform the abstract syntax tree (AST) of the filter set into the corresponding flowchart. This flowchart represents the flow semantics of the filter set. The flow semantics defines how messages/events can flow through the filter set. As the message matching and filtering semantics is not represented by the flowchart, the flowchart does not represent the behavior of a specific message/event in the filter set. To analyze this, the next step is performed.

2. Simulate the execution of the filter set for a specific message/event, using the flowchart for the flow semantics. The message/event matching and filtering semantics are implemented as part of the simulation algorithm. The result of the simulation...
is a state space of the execution of the message; it contains all reachable execution states for the given message. An execution state is the combination of the program counter (pointing to the current location of execution) and the properties of the message (which can change during the execution of the filter set). As the filter set can contain conditional branching (other than through property matching), the resulting state space can contain branching. Each path in the state space represents one possible execution of the message/event.

For certain applications it is necessary to analyze all possible execution paths through the filter set, independent of the specific message/event. All possible execution paths can be obtained by simulating the filter set for all possible messages/events. However, because there are an infinite amount of possible messages/events, this is not possible. Still, all possible execution paths can be determined by taking the following steps:

1. Equivalence classes of messages/events that have the same behavior in the filter set are identified by analyzing the matching parts of filters.
2. For each equivalence class a representative message/event is selected.
3. The filter set is simulated for each representative message.
4. The resulting state spaces are combined into one state space, representing all possible executions of the filter set.

### 7.2. Constructing event equivalence classes

We have previously introduced an event model (Table 1) for physical model instances. To simulate the behavior of a filter set for all possible events in this event model, we need to construct equivalence classes of events. This section explains how such equivalence classes can be created.

#### 7.2.1. Creating equivalence classes for event properties

The first step to create the equivalence classes for events is to create the equivalence classes for each property of the events. The method to create the equivalence classes for a given property depends on the type of the property. Table 2 shows the types of the different event properties. The method to create the equivalence classes for each property type is explained next.

#### 7.2.2. String type

Properties of this type contain String values. An example is the property `variableName`. Matching is performed using String equivalence comparison, e.g., `event.variableName='Tph'`. The String value, e.g., ‘Tph’, to which the value of the property is compared in a given matching part always gives different behavior in the filter set than other String values. The reason for this is that it matches in the given matching part, while other values do not match. We call such a String value a distinguishable value. As the behavior of this value in the filter set is different from the behavior of all other possible String values in the filter set, this value represents an equivalence class. The set of all distinguishable values for a given property `p` and filter set `fs` is represented by `Dist_p^fs`. There is one more equivalence class, which contains all values that are not applied in a matching part to compare the property's value against. As there is no matching part that matches these values, their behavior in the filter set is the same. We use the symbol `e` as the representation for this equivalence class. The set of all equivalence class representatives for the given property `p` and filter set `fs` can be created as follows: 

\[
Repr_p^fs = Dist_p^fs \cup \{e\}.
\]

#### 7.2.3. Enumeration type

Certain properties are of an Enumeration type, such as property `eventType`. The finite set of all possible values for the Enumeration type of property `p` is represented by `T_p`. Each value in `T_p` for which there is a matching part that selects based on this value forms an equivalence class, as its behavior is different from the behavior of the other values in `T_p`. Such a value is again called a distinguishable value. For example, if there is a matching part `event.eventTypes='Change'`, then the behavior of the value ‘Change’ for the property `eventType` is different from the behavior of all other possible values for the property `eventType`. Therefore, the value ‘Change’ forms one equivalence class.

Besides the equivalence classes represented by all distinguishable values `Dist_p^fs`, there is one other equivalence class: the set containing all values of the `Enumeration` type for which there is no matching part that selects based on that value. This set is empty if all possible values of the `Enumeration` type are applied in a matching part. Thus, the set of all equivalence class representatives for the given property `p` and filter set `fs` is defined as follows:

\[
Repr_p^fs = \begin{cases} 
Dist_p^fs \cup \{e\} & \text{if } Dist_p^fs \neq T_p, \\
T_p & \text{otherwise}.
\end{cases}
\]

#### 7.2.4. Numeric type

For Numeric types, matching is performed by testing the equality or inequality of the property with a given value. This value is a distinguishable value. An example is testing the margin property, for example `margin > 2`. In this example, 2 is a distinguishable value. To create the equivalence classes for Numeric types, we make no assumptions about the matching operators; this simplifies the construction of equivalence classes. The equivalence classes for Numeric types are created in the following way:

- Create the ascendingly ordered list of all distinguishable values for the given property in the given filter set, with each value occurring only once in the list.
- Suppose this list contains the following `n` elements: `el_1, el_2, ..., el_n`. The set of equivalence classes `EC_p^fs` contains these elements and the intervals between each element, as follows:

\[
EC_p^fs = \{(-\infty, el_1), (el_1, el_2), (el_2, ... , el_n), (el_n, \infty)\}
\]

The reasoning behind this is that around a distinguishable value the behavior of the filter set differs, as matching is performed using the distinguishable value. Because we make no assumptions about the type of matching (e.g., equality, inequality, greater than), we need to assume that the set containing all values smaller than the distinguishable value, the set containing the distinguishable value and the set containing all values larger than the distinguishable value.

---

2 However, taking the matching operator into account could reduce the number of equivalence classes, making filter reasoning more efficient. Therefore, taking the matching operators into account is considered to be future work.

1 The standard interval notation as defined by ISO 80000-2 [30] is applied.
able value all have different behavior in the filter set, so all are equivalence classes. Because in general there are multiple of these distinguishable values, this creates equivalence classes for each interval between each consecutive distinguishable value.\footnote{Note that this actually creates pseudo-equivalence classes, which means that there might be two equivalence classes that result in the same behavior of the filter set. This is caused by the fact that we do not take the type of matching into account. The disadvantage of pseudo-equivalence classes is that it results in redundant states in the generated state space by the filter reasoning procedure. Redundant states have no effect on the outcome of filter reasoning; they only make filter reasoning less efficient. It is possible to take the type of matching into account, but this results in a less straightforward way to create the equivalence classes.}

The set of representatives $Repr^E$ contains all distinguishable values and one value from each interval. For example, if

$$EC_p^E = \{(-\infty, 1), (1,2), (2, \infty)\}$$

then one possible $Repr^E$ is: \{0.5, 1.5, 2, 2.5\}.

\subsection*{7.2.5. String → numeric type}

The property values is a mapping of String values to Numeric values. Matching is performed by testing the equality or inequality of the values property, indexed with a certain String key, with a given value. An example of a matching statement is $event.values[\text{sensor}] < 1.5$, in which the key is ‘sensor’ and the value is 1.5. The set of equivalence classes for the values property is created as follows.

- First, a list containing all distinguishable keys in the filter set is constructed. This list, containing $n$ elements, is represented as: $Keys = [key_1, key_2, \ldots, key_n]$.
- For each distinguishable key $key_i$, the distinguishable values are identified. These are the values in matching parts that match against the property $event.values[key_i]$. From this set of distinguishable values, the set of equivalence class representatives is created in the same way as with a Numeric type. The set of representatives for $key_i$ is represented by $Repr_{values(key_i)}^E$.
- The set of representatives for the equivalence classes for the values property is constructed using the following equation:

$$Repr^E_{values} = \{\{(key_1, val_1), (key_2, val_2), \ldots, (key_n, val_n)\} | \text{for } i = 1, \ldots, n : val_i \in Repr_{values(key_i)}^E\}$$

For example, if $Keys = \{a, b\}$, $Repr_{values(a)}^E = \{1, 2\}$ and $Repr_{values(b)}^E = \{5, 6\}$, then:

$$Repr^E_{values} = \{\{(a, 1), (b, 5)\}, \{(a, 1), (b, 6)\}, \{(a, 2), (b, 5)\}, \{(a, 2), (b, 6)\}\}$$

\subsection*{7.2.6. Constructing the event equivalence classes}

As the event is defined by the valuation of its properties, the set of equivalence classes for events is the Cartesian product of the sets of equivalence classes for all properties. The set of representative values for the event equivalence classes can therefore be created by taking the Cartesian product of the sets of representative values for all properties:

$$Repr^E = \prod_{p: \text{properties}} Repr^E_p$$

For each element in $Repr^E$ the filter set is simulated. The resulting state spaces are combined to create the state space with all possible executions.

\subsection*{7.3. Applications of composition analysis through filter reasoning}

Filter reasoning has a number of applications. In this subsection we summarize some important applications related to the composition of physical models with GPL modules.

\subsection*{7.3.1. Detecting conflicts in the filter set}

A first application of filter reasoning is to check the consistency of the filter set. For example, there can be unreachable filters in the filter set, filters that never accept, etc. Using the state space that contains all possible executions of the filter set, these conflicts can be detected and their cause can be analyzed\cite{29}.

\subsection*{7.3.2. Detecting redundancy in the physical model}

There can be multiple ways to determine the value of a certain physical variable in a physical model. One source of values for physical variables is from the filter set execution of a Request event. Using filter reasoning it can be determined for which physical variables a Request event results in a value being returned. This information is used to determine which physical variables have multiple sources for their values.

\subsection*{7.3.3. Detecting inapplicable event matching}

For physical variables that do not have multiple sources for their values, it does not make sense to do Inconsistency event matching, as single values cannot be inconsistent. To detect whether Inconsistency event matching is performed for such physical variables, one could suggest performing straightforward analysis of the matching parts in the filter set. This means checking whether there are matching parts that have the selection statement $event.variableName==\text{v}'$ & $event.eventType==\text{'Inconsistency'}$ for all variables $\text{v}$ that do not have multiple sources for their values. But this basic analysis is not sufficient, as composition filters have the capability to change the values of the event properties. Some filter might change the $variableName$ property of the event, which corrupts the results of the basic analysis.

Filter reasoning can be applied to perform this type of analysis correctly, as the simulation of the filter set takes changes of the values of properties into account. To perform this analysis, the following subset of all representative events is taken: the subset that only contains those events for which the $variableName$ property is a physical variable with a single source for its value. The filter set is simulated for all events in this subset. The resulting state space can be checked for whether there exists an $event.eventType==\text{\textquoteleft Inconsistency\textquoteleft}$ check that matches. If this is the case, the property that no inconsistency matching is performed for variables that do not have multiple sources for their values is violated.

\subsection*{7.3.4. Diagnosing faults in the physical model}

Sections 8 and 9 explain, respectively, the runtime verification and diagnosis applied on physical models. Runtime diagnosis leads to the suspicion of an incorrect request value (i.e., a value that is provided by the composition filters after a Request event). In this case, the filter set can be analyzed further to determine the origin of the value, e.g., a call to a sensor component. Hereby, the behavior of the filter set is simulated for the given event to detect the single origin or the multiple possible origins of the value.

\subsection*{7.3.5. Efficient execution of composition filters}

The current implementation of our approach makes use of the Compose* interpreter to execute defined composition filters when an event occurs in the physical model instance. Such an interpreter based approach provides the flexibility to experiment with the implementation, but it does not optimize for efficiency. Instead,
it introduces a runtime performance overhead. To apply the approach in industry, more efficient execution is necessary.

More efficient execution of composition filters can be obtained by compiling the filter set to GPL code [29]. The compilation can be performed in three main steps:

- Locate the artifacts (e.g., classes, physical model instances) on which composition filters are superimposed.
- For each of these artifacts, find the locations in their code at which a message is sent or an event occurs (in aspect-oriented programming terminology, these are called join-point shadows [31]).
- For each of these locations, apply filter reasoning on the filter set with the message or event corresponding to the code location. The result of this filter reasoning is the specific behavior of the filter set for the given message or event. Code can be generated that executes this specific behavior and this code can be woven at the specific location in the artifact.

In our approach, the composition filters are superimposed on physical model instances. Code generation for the composition filters should be performed together with code generation for the physical model instance, as the generated GPL code with filter behavior should be woven into the generated GPL code of the physical model instance.

8. Runtime verification of physical models

To verify physical models at runtime, we need to check whether these physical models correspond to actual physical reality. To do this, we generalize the techniques used for state observers [7] in control engineering. State observers are implementations of physical models that provide more information about the system's state than is available from sensors. They are kept consistent with the system's state by calibrating the model's state with (redundant) information known from sensors [7].

State observers are one application of physical model instances. Redundancy can be present in physical model instances. In this case redundancy means that certain physical variables in the physical model have multiple sources for their values. Examples of such sources are:

- Additional sensors to determine the value of a physical variable, besides an already existing relationship in the physical model to calculate the value for the same physical variable. This is a situation that is rare at system deployment, as the physical models and the physical relationships within the models are introduced because there is no available sensor information. One application of such sensors is to calibrate the physical state in the physical model instance, for which the sensor only occasionally gives a reading. The Drum Shutting case study contains such a sensor.
- Redundancy in the physical relationships in the physical model. If there are multiple physical relationships in the physical model that calculate the value of the same physical variable, the results can be compared.

As an example, consider an extension of the Warm Process case study that adds redundancy to the BeltTemperature model. Suppose that the BeltTemperature model in the Warm Process case study also contains the following equation:

$$T_{belt} = c5 \cdot (T_{contact} + c6 \cdot T_{ph}) \cdot \sqrt{v}$$

This equation determines the temperature at the sensor location ($T_{belt}$) from other physical variables ($T_{contact}$, $T_{ph}$ and $v$). Listing 7 shows the resulting SIDOPS+ specification of the BeltTemperature model.

The description of the Warm Process case study in Section 3.1 shows that there is also a sensor to measure the belt temperature. Listing 8 shows the composition filters specification that retrieves the sensor value when there is a Request event.

In this example, there are multiple sources for the value of $T_{belt}$: the Eq. (5) and the composition filter that returns a sensor reading after a Request event. Fig. 6 shows the dependency graph corresponding to this example. This graph clearly shows multiple incoming edges to the $T_{belt}$ node.
8.1. Detecting redundancy in the physical model

Redundancy is easily recognizable in dependency graphs; there is a redundant calculation if a variable node has multiple incoming edges. This means that the corresponding variable can be calculated in multiple ways. The dependency graph in Fig. 6 contains one redundant calculation. The variable $T_{belt}$ can be determined in two ways; it can be derived from a sensor input and through Eq. (5). Note that there is actually a cycle in the graph, in which $T_{belt}$ depends on itself through Eqs. 2 and 5.

8.2. Checking the consistency of the physical model

Analogous to state observers, the redundancy in the physical model instance can be utilized to verify whether the corresponding physical model is consistent with physical reality. The physical model is consistent if all the ways to calculate the value of the variable result in the same value. If not all results are equal, one of the value sources is not correct. The Evaluator component, as part of the interpreter, evaluates all possible sources of values for the physical variables. It also checks whether the multiple outcomes are consistent. If not, it issues an Inconsistency event.

8.3. Handling inconsistency

To handle inconsistencies in the physical model, composition filters can be specified to filter and match the issued Inconsistency events of $T_{belt}$. The type of the filter is Logging, which means that it performs a logging operation when it matches.

8.4. Coping with deviation in outcomes

In reality, there are often small deviations between the outcomes, because of small error-margins in the sensors or formulas, for which the system has been designed to be robust. Therefore, to do monitoring and checking, such error-margins should be taken into account (i.e., if the difference between the redundant values is within a certain safe range, there is no indication of an error).5

The event model introduced in Section 6.2 provides means to ignore small deviations by using the margin property. The example in Listing 9 demonstrates this property; it only matches for Inconsistency events of $T_{belt}$ if the margin is larger than 0.1.

9. Fault diagnosis

When the different values for a physical variable are inconsistent, this indicates that there is an error. The next step is diagnosing the cause of the error. There are multiple possible causes for errors, e.g., a malfunctioning sensor or actuator, a failing physical component or a fault in the implementation of the physical relationship in the model. Possible causes can be determined using the derivation graph, which is defined as follows.

**Definition 1 (Derivation Graph).** A derivation graph of a physical model is a directed graph that reflects how the values of the physical variables in the physical model are derived from the values of other physical variables by the evaluation algorithm.

The derivation graph is different from the dependency graph; the dependency graph reflects the relationships between the physical variables as specified by the SIDOPS+ specifications. Due to the backward solving step in the algorithm, the actual derivation order may be different. Listing 10 shows an example SIDOPS+ specification and Fig. 7 shows the corresponding dependency graph (the numbers in the equation nodes correspond to the ordering in the listing).

The derivation graph as shown in Fig. 8 is obtained by traversing the dependency graph. In the example, the value of $v6$ is determined using backward solving from equation eq3 and the values of $v3$ and $v7$. This is shown in the derivation graph by the change of direction of the edges $(v7,eq3)$ and $(Eq. 3,v6)$. The value of $v5$ is also determined using backward solving, as is shown in the derivation graph by the change of direction of the edges $(v6,eq2)$ and $(Eq. 2,v5)$. The bi-directional edges attached to eq8 in the dependency graph have been replaced in the derivation graph by unidirectional edges that indicate that the value of $v13$ is derived from the values of $v5$, $v7$ and $v11$.

The derivation graph can be used to diagnose faults by tracing back all paths from the variable that has inconsistent sources for its value. Systematic fault diagnosis starts in the derivation graph at the variable node that corresponds to the inconsistent physical variable ($v2$ in the example). In a systematic way, the derivation graph is traversed backward, where each type of graph structure encountered provides fault diagnosis information. Table 3 shows the different possible graph structures that can be encountered. In black, the encountered graph structure is shown. In gray, the graph structure that is traversed next is shown. The fault diagnosis...
information that can be derived from each graph structure is explained next.

9.1. Inconsistent variable node

This is the starting point of the diagnosis. This structure leads to the following diagnosis information:

- For each of the incoming edges: the value provided by the edge might be incorrect.

9.2. Request value edge

The request value edge leads to the following fault diagnosis information:

- The outcome of the filter evaluation might be incorrect.

At this point, the traversal of the derivation graph stops. But fault diagnosis could continue with the filter set, deriving the source of the value, e.g., a module that provides a sensor reading. This could mean that the module does not work correctly, or the sensor is broken. Section 7.3 explains how this can be done by analyzing the filter set.

9.3. Set value edge

The set value edge leads to the following fault diagnosis information:

- The provided value might be incorrect.
This is also an end-point of the traversal of the derivation graph. GPL code analysis could be done to determine the source of the provided value.

9.4. Equation result edge

The equation result edge leads to the following fault diagnosis information:

- The outcome of the equation might be incorrect.
- The traversal continues with the corresponding equation node.

9.5. Variable value edge

The variable value edge leads to the following fault diagnosis information:

- The value of the variable might be incorrect.
- The traversal continues with the corresponding variable node.

9.6. Equation node

The equation node leads to the following fault diagnosis information:

- The equation itself might be incorrect.
- For each incoming edge: the value on the incoming edge might be incorrect.

If the equation node has been visited before, traversal of the graph ends. Otherwise, the traversal continues to all incoming edges.

9.7. Variable node

The variable node leads to the following fault diagnosis information:

- For each incoming edge: the value on the incoming edge might be incorrect.

If the variable node has been visited before, traversal of the graph ends. Otherwise, the traversal continues to all incoming edges.

Note that a variable node has multiple incoming edges only if it has multiple sources of information. As the traversal algorithm cannot detect from the derivation graph which source of value is used as the output value, it has to traverse all incoming edges. Such information could be maintained while traversing the dependency graph for constructing the derivation graph. This can make the diagnosis process more focused, as only the single edge that is the source of information needs to be analyzed.

As an example for the application of structured fault diagnosis, consider the Warm Process case study. Suppose that the sensor reading for variable \( T_{belt} \) gives a different value than the evaluation of Eq. (5). The dependency graph shown in Fig. 6 is in this case also the derivation graph. This derivation graph is used for the structured fault diagnosis, leading to the following diagnosis information:

1. The request value edge might be incorrect: this may indicate a malfunctioning \( T_{belt} \) sensor.
2. The outcome of Eq. (5) might be incorrect. This may indicate:
   (a) Eq. (5) itself might be incorrect.
   (b) The value of \( T_{ph} \) might be incorrect.
      i. The request value edge might be incorrect: This may indicate a malfunctioning \( T_{ph} \) sensor.
   (c) The value of \( v \) might be incorrect.
      i. The request value edge might be incorrect: This may indicate a problem in the actuation/controlling of \( v \), either in software or in hardware.
   (d) The value of \( T_{contact} \) might be incorrect. \( \rightarrow \) the outcome of Eq. (2) might be incorrect.
      i. Eq. (2) itself might be incorrect.
      ii. The value of \( T_{belt} \) might be incorrect. We already encountered the \( T_{belt} \) variable node (there is a cycle in the graph \(^6\)), so evaluation stops here.
      iii. The value of \( P_{rad} \) might be incorrect.
         A. The request value edge might be incorrect. This indicates a problem in the actuation/controlling of the radiator, either in software or in hardware.
         iv. The value of \( v \) might be incorrect. This case has already been taken into account in point 2c.

\(^6\) Such cyclic dependencies could be detected to provide a warning for potential faults. However, they are not necessarily harmful, as proven by the introduced example case.
10. Evaluation of the approach

This section evaluates the effectiveness of the approach, by demonstrating several scenarios in which it can be applied. These scenarios illustrate the diverse situations, in which the approach is capable of detecting different type of errors. Of course, the effectiveness of the approach in a given scenario depends on the amount of redundancy in the physical model; the more redundant information available, the more errors can be detected.

10.1. Detecting inconsistencies at runtime

Inconsistencies in the values for physical variables may indicate that the physical model does not correspond to physical reality. Composition filters can be implemented to monitor for inconsistencies in the physical variables that have a redundant value source. The aspect-oriented features of composition filters make it possible to define a single filter that monitors and handles all inconsistencies. Listing 11 shows an example of such a composition filter specification. Line 3 shows the definition of a `Logging` filter that matches all `Inconsistency` events that have a significant margin. All occurrences of these events are logged. Lines 6–11 show that the filter module is superimposed on all physical model instances in the system.

10.2. Monitor wear and tear

One advantage of the runtime verification approach is that it becomes possible to monitor for wear and tear in the system. We show this using the Warm Process case study.

The printing system in the Warm Process case study contains a radiator component to heat the toner belt. Over time, the radiator component may get polluted and therefore less efficient. If not detected early enough, this can cause damage to the system.

Detection of radiator pollution has been implemented using the `Belt Temperature` physical model given in Listing 7. This model contains redundancy; the physical variable \( T_{belt} \) has two sources for its value: a sensor and the added equation. With a clean radiator, the two sources provide the same value. But when the radiator gets polluted, the two values begin to differ. This can be detected by the composition filter specification shown in Listing 12. Lines 5 and 6 show the `radWear` filter. This filter matches `Inconsistency` events of \( T_{belt} \) with a margin larger than 0.2. If the filter matches, it dispatches the message `serviceCall` to the `ServiceModule` in the GPL.\(^7\)

10.3. Monitor acceptable ranges

Besides monitoring for inconsistencies, the Composition Filters model also provides means to perform other runtime verification tasks within physical models. One example is monitoring whether a physical variable stays within an acceptable range. An example of this is shown in Listing 13. The specific physical variable in this example is \( T_{ph} \). The `eventType` is `Change`, reflecting the fact that checking is done when the value of the variable changes. The acceptable range for the physical variable is between 60 and 100. If the value is outside this range, the composition filter matches and the event is logged.

10.4. Calibration

Inconsistencies in multiple values of a physical variable do not always indicate a failure. This section shows an example in which inconsistency monitoring is used to calibrate the physical state in a physical model instance. However, precautions are taken to cope with a broken sensor, a scenario that can happen in practice. The implementation of these precautions using our approach are explained in the next subsection.

In the Drum Shuttling case study, the z-position of the drum (\( z_{Pos} \)) is derived from the step position of the stepper motor (\( step_{Pos} \)). This step position is updated if a step is actuated. However, the stepper motor occasionally fails to perform a step when it is actuated, creating a deviation between the physical model and

\(^7\) Note that wear and tear might not be the only cause for the inconsistency; the full diagnosis process of the inconsistency has been outlined in Section 9.
physical reality. Small deviations do not cause a problem. However, when the physical model is not corrected once in a while, the errors may accumulate into larger deviations. To cope with this problem, there is a calibration sensor in the system, called home sensor, that is triggered when the drum reaches a specific z-position. Example Section 10.4.1 shows how the home sensor is used to calibrate the physical model instance.

### 10.4.1. Example: calibration of a physical model instance

#### 10.4.1.1. Software design

Fig. 9 shows the software design of the Drum Shuttling case study. It shows two instances of the \( \text{(stepperMotor, cam)} \) physical model. The first instance is used to provide a setpoint value for the step position to the StepperController. The second instance is used to perform calibration of the current step position.

#### 10.4.1.2. Composition definition

In the Shuttling2 physical model instance, the value of the physical variable stepPos is set by the StepperController module after each performed step, using the base interface of the physical model instance.

*Listing 14* shows the composition filters specification to calibrate Shuttling2. There are three filter modules. The zPosRequestHandler filter module handles Request events for zPos. But it only accepts if the condition isActive is true. This condition is defined to check whether the home sensor is active. If the filter matches, the result of the Request event is set to home position (which is defined as the constant C_homePosition).

The zPosMonitor filter module implements the actual calibration. It monitors for Inconsistency events in zPos and selects the value that is provided by the homingResult filter (i.e., the home position). It also enforces this result upon the entire physical state, by setting the enforceResult property to true.

The zPosLogging filter module performs logging of the inconsistency if the deviation is larger than two, for diagnosing the problem later on.

Lines 20 until 25 show how the two filter modules are superimposed on the module ShuttlingModel.

#### 10.5. Broken sensor or stepper motor detection

The previous example showed how the home sensor is used to calibrate the physical model instance. This works fine when both the home sensor and stepper motor work properly. But if one of them is broken, this leads to incorrect results, which may even damage the physical system. For example, when the sensor is broken and always gives a signal, the physical model instance will be continuously calibrated to the home position. This continuous calibration means that stepPos remains the same. The value of stepPos, as indirectly requested by the ShuttlingController, will continue to move further away from the home position. So, the difference between stepPos and stepPos in the model increases, and this difference is larger than in reality, as in reality stepPos does change and as such is closer to stepPos. That the calculated difference between stepPos and stepPos is larger than in reality means that the StepperController will perform more steps to bring stepPos to stepPos than actually needed. Eventually, the stepper motor will be controlled in such a way that the zPos of the drum will go outside its physical boundaries, which possibly results in damage of the physical system. Using proper runtime verification, such problems can be prevented. In this subsection we show how the detection of a malfunctioning sensor or malfunctioning stepper motor can be implemented using the techniques presented in this chapter. A malfunctioning sensor can give two results: either the sensor never gives a signal or the sensor always gives a signal. Example Section 10.5.1 shows how a sensor that...
never gives a signal can be detected. Example Section 10.5.2 shows how a sensor that always gives a signal can be detected.

10.5.1. Example: detection of never active sensor

Detection of a sensor that is never active is performed as follows. When the drum is supposed to be at the home position ($zPos = C_{homePosition}$), and the sensor is not active, then detection starts. If the sensor is working properly, an active signal will arrive within a limited number of steps. But if the sensor is broken, an active signal will never arrive. Therefore, if the $zPos$ has changed outside certain boundaries around $C_{homePosition}$ and still there is no signal, the sensor is assumed to be broken. These boundaries are represented by the constants $C_{checkLowBound}$ and $C_{checkUpBound}$. If $zPos$ is still within the boundaries and a signal of the sensor does arrive, checking for an inactive sensor stops. Note that when an always inactive sensor is detected, this means that either the sensor is broken and gives no signal or that the stepper motor is broken and the drum never reaches the home position. Listing 15 shows the specification of a filter module that detects inactive sensors.

The filter module uses an internal object of class Detector, to maintain state about the detection. There are three filters. The filter startCheck starts the detection of an inactive sensor. It

```java
filtermodule zPosRequestHandler{
  externals
  homeSensor: Sensor = IO.getShuttlingHomeSensor();
  conditions
  isActive: homeSensor.isActive();
  outputfilters
  homingResult: Result = (event.variableName == 'zPos' &
    event.eventType == 'CheckUpdate' & isActive)
    {event.returnValue = C_homePosition;
     event.returnIdentifier = 'homeSensor';};
}

filtermodule zPosMonitor{
  outputfilters
  incomHandler: Result = (event.variableName == 'zPos' &
    event.eventType == 'Inconsistency')
    {event.returnValue = event.values['homeSensor'];
     event.enforceResult = 'true';};
}

filtermodule zPosLogging{
  outputfilters
  zPosLog: Logging = (event.variableName == 'zPos' &
    event.eventType == 'Inconsistency' & event.margin > 2);
}

superimposition{
  selectors
  models = { M | isModelInstance (M, [Shuttling2]) };
  filtermodules
  models <- zPosRequestHandler, zPosLogging, zPosMonitor;}
```

Listing 14. Composition filters specification to handle home sensor for calibration.

```java
filtermodule InactiveSensorFM{
  externals
  homeSensor: Sensor = IO.getShuttlingHomeSensor();
  internals
  detector: Detector;
  conditions
  isActive: homeSensor.isActive();
  checkInactive: detector.checkInactive();
  outputfilters
  startCheck: Before = (event.variableName == 'zPos' &
    event.eventType == 'Change' &
    event.value == C_homePosition & isActive)
    {filter.target = detector;
     filter.selector = 'startCheck';};
  errorDetect: Before = (event.variableName == 'zPos' &
    event.eventType == 'Change' &
    event.value < C_checkLowBound | event.value > C_checkUpBound & checkInactive & !isActive)
    {filter.target = detector;
     filter.selector = 'sensorNoSignal';};
  stopCheck: Before = (event.variableName == 'zPos' &
    event.eventType == 'Change' & isActive)
    {filter.target = detector;
     filter.selector = 'stopCheck';}
}
```

Listing 15. Composition filters specification to detect inactive sensor.
matches when a change of zPos to ChomePosition occurs and the sensor
is not active. When the filter matches, it performs a call to startCheck in Detector to set the condition checkInactive to true. The filter errorDetect detects that zPos is outside the boundaries and we are still waiting for a signal. This filter executes error code in Detector. The filter stopCheck detects that the sensor is active and stops the checking by calling stopCheck in Detector. This call sets the condition checkInactive to false.

10.5.2. Example: detection of always active sensor

Detection of a sensor that is always active is performed as follows. The StepperController actuates the stepper motor to make steps. If everything works properly, then if the drum is at the home position and a step is made, the active sensor has to become inactive. The only exception to this property is when the stepper motor accidentally misses a step; in this case the sensor is still active as the drum is still in home position. But if multiple steps are made, for example more than 4, and the sensor is still active, this is assumed not to be caused by the expected behavior of the stepper motor to miss a step once in a while, but by a broken sensor that always gives a signal. Note that a sensor that is always active may also indicate a broken stepper motor, while the drum is exactly at the home position. Listing 16 shows the specification of a filter module that detects always active sensors.

There are two filters. The filter doCheck matches when stepPos is updated and the sensor is active. The filter action is a call to method activeDetect in class Detector. This method increments a counter that indicates how many times after each other a step is made while the sensor remains active. The method also implements the check to detect that the counter becomes to large and the method implements the handling of the detected failure. The condition checkActive is true when the counter is larger than 0, otherwise the condition is false. The filter resetCheck matches when stepPos is updated and the sensor is not active. A call to the method inactiveDetect is performed to reset the counter in Detector to 0.

11. Discussion

This section discusses some additional subjects related to our approach and the implementation of our approach.

11.1. Runtime verification performance

Applying runtime verification means that additional behavior is executed, leading to a certain performance overhead. We used an interpreter based implementation for the evaluation of the SIDOPS+ specifications and the execution of the composition filters for monitoring. Such an interpreter based approach introduces considerable runtime overhead. However, the aim of the interpreter based implementation is to experiment with and demonstrate our approach, not to provide an efficient runtime environment. Efficient compilation algorithms for aspect-oriented languages, such as the Composition Filters model, are known in literature, e.g., in [32,6,29].

11.2. Size of the state-space

In [29] the complexity of the general approach for filter reasoning has been investigated. A similar analysis can be applied to the adapted filter reasoning approach presented in this paper. Following the reasoning given in [29]:

\[ O(\#states) = O(\#flownodes \cdot \#message_equivalence_classes) \]

\[ O(\#flownodes) \text{ is linear in the size of the filter set, expressed as the number of matching parts in the filter set, as explained in [29], so} \]

\[ O(\#flownodes) = O(\#matchingParts) \]

The number of event equivalence classes is the product of the number of equivalence classes for each property (of the eight properties variableName, eventType, value, valueReturn, Value, returnIdentifier, enforceReturn). Two of these properties (eventType and enforceReturn) have Enumeration type, which have respectively 3 and 2 equivalence classes. Five of the six other properties have String and Numeric types, which have worst case size O(\#matchingParts). The property values has type String → Numeric. The number of equivalence classes for this property is worst case O(\#matchingParts^2). So, the size of the set of event equivalence classes is worst case:

\[ O(\#states) = O(\#flownodes \cdot 3 \cdot 2 \cdot \#matchingParts^2) \]

As the number of matching parts is O(\#flownodes), the previous equation can be simplified into:

\[ O(\#states) = O(\#flownodes^2) \]

So, the space complexity is worst case polynomial, i.e., to the power of 8 in the size of the filter set. But as explained in [29], and as encountered in practice, filter sets are usually small, and it will be rare for event properties having more than a dozen equivalence classes. This means that the size of the state space remains manageable for most applications. For exceptionally large filter sets, we can

---

Listing 16. Composition filters specification to detect inactive sensor.

```
filtermodule ActiveSensorFM{
  externals
  homeSensor: Sensor = IO.getShuttlingHomeSensor();
  internals
  detector: Detector;
  conditions
  isActive: homeSensorisActive();
  checkActive: detector.checkActive();
  outputFilters
  doCheck: Before = (event.variableName==‘stepPos’ &
    event.eventType==‘Update’ & isActive)
    {filter.target=detector;
      filter.selector=‘activeDetect’;};
  resetCheck: Before = (event.variableName==‘stepPos’ &
    event.eventType==‘Update’ & checkActive & !isActive)
    {filter.target=detector;
      filter.selector=‘inactiveDetect’;}
}
```
think of a number of possible optimization techniques to reduce the number of equivalence classes for messages. These are actually pseudo-equivalence classes, as explained in Section 6; with smarter analysis a lot of the duplicate pseudo-equivalence classes can be removed. This is considered to be a future work.

11.3. Size of the dependency graph

The dependency graph is used to diagnose faults in the physical model. For efficient diagnosis, the size of the dependency graph should be manageable. The dependency graph consists of two different types of nodes: variable nodes and equation nodes. The number of variable nodes in the dependency graph is equal to the number of variables in the physical model (e.g., as specified in a SIDOPS+ specification). The number of equation nodes is equal to the number of equations in the physical model. The number of edges in the dependency graph is equivalent to the number of variable occurrences in the equations of the physical model. The number of variable occurrences is linear in the size of the physical model.

So, we can conclude that the size of the dependency graph is linear in the size of the physical model, which is specified in a SIDOPS+ specification. As such, efficient fault diagnosis can be performed using a dependency graph.

11.4. Moment of checking

A fault or inconsistency in a physical model can lead to a failure when the physical model is evaluated and the results of this evaluation are used by GPL modules. Since we want to monitor whether there are inconsistencies in the physical model that can lead to failures of the system, the best time to do monitoring is after the physical model has been evaluated but before the result is used. In this way, the results of the evaluation can be used by the monitoring code, reducing the overhead of performing additional calculations. Furthermore, action can be taken before erroneous results of the evaluation are used by GPL modules and lead to failures in the system. This moment of checking is enforced by the evaluation algorithm: first the inconsistencies are resolved before Change events are communicated.

One could argue that monitoring can also be performed at other time instances, for example to diagnose problems in an earlier stage. However, this may lead to complications, as the physical relationships in the physical model may be designed to be consistent only when the physical model instance is evaluated. Furthermore, performing monitoring at other time instances reduces performance of the system, as additional calculation and solving of the physical relationships in the physical model needs to be performed, to derive values when the physical model instance is not evaluated.

11.5. Recovery actions

When an inconsistency has been detected, a recovery action can be taken. Depending on how the engineer perceives the severity of the inconsistency, there are several options, which include:

- Stop the operation of the system, for safety-critical operations.
- Select one of the calculated values, e.g., randomly, based on a voting scheme or based on a preference.
- Log the inconsistency for diagnosis.

In some cases, the system can also be reconfigured at runtime if it is necessary to recover from an error. For example, Babty et al. [33] facilitate dynamic reconfiguration of systems for error recovery.

12. Related work

In the following subsections, we provide a review of the related work regarding several approaches introduced in this paper.

12.1. Composition analysis

In our approach, physical models are composed with GPL modules using the aspect-oriented Composition Filters model. The composition filter specifications are analyzed to detect errors and inconsistencies at design time. Krishnamurthi et al. describe in [34] how aspects can be verified modularly, i.e. independent from the base program. They try to verify that introduced aspects do not invalidate certain enforced properties of the base program. To do this, they use a state machine representation of the aspects. Related to this is work by Goldman and Katz, who introduce in [35] a technique to verify modularly that an aspect satisfies certain properties if the base system satisfies certain assumptions of the aspect. The state space generated with filter reasoning can be used as the state transition diagram used in these verification techniques. Furthermore, the consistency analysis of composition filters can also be performed modularly, without knowledge of the base system (in this case, the physical model instance and GPL modules).

Our filter reasoning approach is able to detect certain consistency conflicts, for example caused by an incorrect ordering of the composition filters. This is one type of conflict that can occur in aspect-oriented programming languages. Different types of conflicts have been investigated in literature. Störzer and Krinke show in [36] how binding interferences in AspectJ, caused by the introduction and hierarchy modification features of AspectJ, can be analyzed. Kniesel describes how to detect and resolve weaving interactions in [37]. Havinga et al. introduced techniques to detect composition conflicts that are caused by introductions [38].

Dürr investigated how to detect behavioral conflicts among aspects [39]. Behavioral conflicts are conflicts in which the behavior of an aspect interferes with the behavior of other aspects, resulting in unexpected behavior. Dürr's approach for behavioral reasoning makes use of a resource-operation model. He applied this approach to the Composition Filters model, using filter reasoning as the basis for the analysis.

12.2. Runtime verification of safety-critical systems

Runtime verification is widely applied for safety-critical systems such as nuclear power plants and transportation systems. We have adopted several techniques that have been previously used in different domains including safety-critical systems. For instance, tracing the dependency graph to diagnose faults is one of such generic techniques. However, there are two fundamental differences between our approach and runtime verification approaches applied on safety critical systems.

First, runtime verification approaches for safety critical software are mainly concerned with checking (the violation of) a set of formally specified safety properties, e.g., “the robot arms do not move horizontally while they are in their down position” [40]. These properties are specified with various formalisms such as message sequence charts [41]. These properties are verified and enforced on the behavior of the system as a whole. In our work, we do not focus on the verification of the system or the control software as a whole by checking certain properties. We aim at verifying the consistency of physical characteristics that are implemented in software (with respect to physical reality). For this purpose, we separate their implementation from the control software.
and check their consistency with respect to observations from the environment.

The second difference is about the cost sensitivity. For safety–critical systems, safety is the primary concern and any failure that can cause harm to people/environment must be prevented. In that context, the additional cost of verification due to the required hardware and software resources is a minor issue. As such, usually processing units are replicated, and fault-tolerant algorithms are used for comparing their outputs [11]. Digital document printing systems are less subjected to catastrophic failures and the cost is one of the primary concerns. In our approach, we utilize only the existing redundancy in the system in terms of available models implemented in the software (e.g., no extra specifications except for the composition of the system) and the available hardware (e.g., no extra sensors).

12.3. Fault diagnosis

Fault diagnosis aims at determining the health state of the system or components in the system, by analyzing the output of the system given a certain input. There are two approaches to diagnose the location of faults in components. Model-based diagnosis, as introduced by Reiter [42] and de Kleer and Williams [43], uses a model of the system to diagnose the failing component based on the system’s input and output. Spectrum-based fault localization is a statistical approach that diagnoses failing components by correlating failures in the output with execution traces [44]. van Gemund et al. combined both approaches to be applied on the combination of embedded system and the corresponding embedded software [45,46].

Our approach supports the model-based diagnosis approach by Reiter and de Kleer in the context of physical models; we can define a physical model primarily aimed at determining values of physical variables that are checked with sensor values, to detect inconsistencies in the sensor values. A difference with the model-based diagnosis approach is that our approach does not only apply models to verify sensor outcomes, but also uses sensor outcomes to verify the physical models and our approach uses the physical models to implement behavior in software. In the last case, runtime verification detects inconsistencies in the physical model, as well as the wear and tear of hardware in the physical system. Such errors are diagnosed using a graph model derived from the physical models, which reveals incorrect physical relationships or to the existence of malfunctioning system components.

We have demonstrated our approach on two industrial case studies for the analysis and runtime verification of digital document printing systems. We have seen that explicit and separate specification of physical models can be automatically composed with the implemented control logic. Aspect-oriented composition can be analyzed at design time for detecting inconsistencies and interference. In addition, monitors are generated to detect faults that are left undetected at design time or, that manifest themselves later at operational time. In addition, these monitors can be used for calibration and for detecting wear and tear of components in the physical system. The complementary static and runtime verification techniques are applied together to increase the reliability of embedded control software.

14. Future work

In this work, we exploited existing redundancy in the physical models to check their consistency. Hence, the types of errors that can be detected are limited based on the amount of redundancy. Our approach can be extended with a methodology, where physical models are designed by systematically evaluating the trade-off between the cost of redundancy and the number of types of errors that can be detected. The approach can also be extended with runtime diagnosis and recovery techniques to not only detect, but also recover from errors (i.e., inaccuracies of the physical models) at runtime.

There might be many different assumptions implemented within a system regarding its environment. Physical models constitute only one aspect of these assumptions. Other examples can be given as certain component models of the hardware or usage models. Our approach can be generalized to cover such aspects as well. In general, an implementation of any assumption regarding the system environment can be verified with respect to the actual environment. The verification is performed at runtime by checking whether the (sensor) measurements from the environment correspond to the expectations of the system based on its assumptions about its environment.

Acknowledgments

This work has been carried out as part of the OCTOPUS project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute program. We thank Jacques Verriet from ESI and Lodewijk Bergmans from our group for reviewing an earlier version of this paper and providing useful feedback.
References


