2011 Fifth International Conference on Secure Software Integration and Reliability Improvement

Runtime Verification of Domain-Specific Models of Physical Characteristics in Control Software

Arjan de Roo, Hasan Sözer, Mehmet Akşit
Software Engineering group
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente, The Netherlands
Email: {roo, sozerh, aksit}@ewi.utwente.nl

***Abstract*—Control logic of embedded systems is nowadays largely implemented in software. Such control software implements, among others, models of physical characteristics, like heat exchange among system components. Due to evolution of system properties and increasing complexity, faults can be left undetected in these models. Therefore, their accuracy must be verified at runtime.**

Traditional runtime verification techniques that are based on states and/or 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. This paper presents a novel approach to explicitly specify models of physical characteristics using a domain-specific language, to define monitors for inconsistencies by detecting and exploiting redundancy in these models, and to realize these monitors using an aspect-oriented approach. The approach is applied to two industrial case studies.

I. Introduction

Many embedded systems today are largely controlled by software. For example, the control of overall system behavior in digital document printing systems is mainly performed by software. The size and complexity of such embedded software systems is continuously increasing due to the demand for new functionality, increasing variety in the types of hardware that need to be controlled and more refined/optimized control. 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.

Fault prevention and removal techniques usually fall short to ensure the reliability of embedded systems, which are developed under time-to-market pressure. For example, software testing is widely used; however, due to time limitations and the sheer size and complexity, testing software exhaustively has become prohibitive. Therefore, it is necessary to detect and cope with residual defects at runtime. As such, runtime verification has to be adopted in embedded systems as a complementary approach to increase the reliability.

An important characteristic of embedded control software is that it includes models of physical characteristics (which we will abbreviate to *physical models* in the following), for example models of the natural relationships between physical variables. These physical models mainly determine the control behavior. For example, several calculations 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 functioning of the system. However, there can be errors in these estimations due to a number of reasons, e.g., *i)* data received from the sensors can be inaccurate, *ii)* there can be undiscovered faults in the implementation, and *iii)* some implementations can become obsolete and invalid when the working context or system properties change (change of hardware, wear and tear of the hardware). Therefore, such estimations must be verified at runtime.

Observers and monitors are essential elements for runtime verification. Traditionally, these are extracted/generated/defined based on specifications of events and states of interest in software and properties on these events/states that need to be verified. However, this approach is inadequate for verifying the physical models implemented in software. The monitors, in this case, are concerned with how physical characteristics are represented in software. This is not directly apparent from the encountered states and events during the software execution. 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 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.

In this paper, we propose a novel approach, where physical models in an embedded system are specified with the domain-specific language SIDOPS+ from the 20-Sim toolset [1], [2]. Such specifications are part of the software and are executed using an interpreter. An approach to verify these physical models, taken from control engineering literature on state observers [3], is to use redundancy in the model (e.g., multiple relationships for the same physical variable, additional sensors). Specified relationships in the domain-specific language define a dependency graph, in which redundant paths reveal the information that can be used to check for consistency. The same information can be utilized for fault diagnosis. Monitors are created using the aspect-oriented composition

978-0-7695-4453-3/11 $25.00 © 2011 IEEE
DOI 10.1109/SSIRI.2011.14

filters language. We apply our approach on two industrial case studies for runtime verification of digital document printing systems.

To summarize, our paper provides the following contributions:

- The application of a domain-specific language to explicitly specify physical models in software for the purpose of verifying their correctness.
- A method to detect redundancy in the physical models and utilize this to verify their consistency with physical reality.
- An aspect-oriented approach to create monitors for verification and handling of inconsistencies in the physical models.

The remainder of this paper is organized as follows. Section II motivates our research, including an industrial case study to illustrate the problems and challenges addressed by this paper. Our approach is explained in Section III. Section IV presents implementation details of our approach. In Section V we apply our approach to a second industrial case study. Several aspects of our approach are discussed in Section VI. Related work is discussed in Section VII. In Section VIII a conclusion is given and future work is outlined.

II. Problem Statement

In this section we illustrate that physical models are part of control software, we motivate why such physical models need to be verified at runtime and we explain a number of relevant challenges that arise when we want to perform this verification.

A. Physical Models Implemented in Control Software

This section describes an industrial case study that shows that physical models are being implemented in control software. This case study will also be used as a running example to explain our approach. Although this case is simplified for presentational purposes, it is still relevant and sufficient to illustrate the problem and our solution approach. In fact, within the context of the Octopus project[1] [4], our approach has been applied in practice. The case focuses on the *warm process* subsystem of a printing system, which is responsible for transferring a *toner image* to paper.

Figure 1 gives a schematic view of the components in the printing system responsible for the warm process. The warm process has two main parts; a *paper path* for transporting sheets of paper and a *toner belt* for transporting toner images. For correct printing, both the paper as well as 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 paper and a *radiator* to heat the toner belt.

[1]The Octopus project is a joint effort in a consortium of both industrial and academic partners: Océ-Technologies B.V. (one of the world's leading manufacturers of printer and copier systems), the Embedded Systems Institute and four Dutch universities [4].

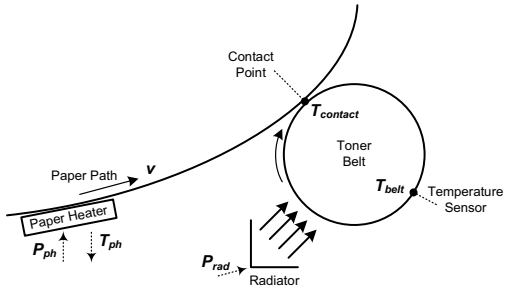

Fig. 1. Schematic view of the warm process

Software has been implemented to control the heaters to maintain the required temperatures for correct printing. Figure 2 shows the software structure and the data-flow between the different modules.

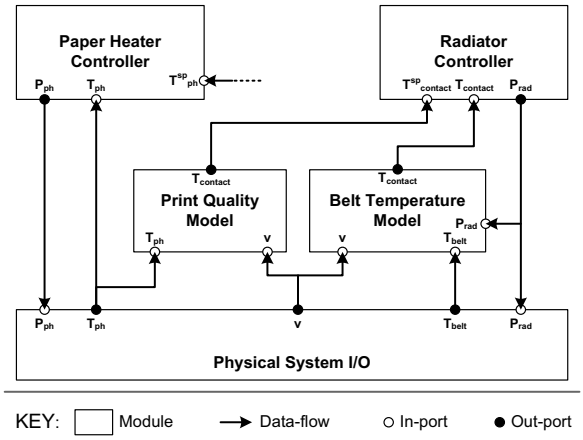

Fig. 2. Schematic overview of the software structure

The `Physical System I/O` module provides an interface to the following sensors and actuators of the system:

- T_{ph}: Sensor that measures the temperature of the paper heater.
- T_{belt}: Sensor that measures the temperature at the sensor location on the toner belt.
- v: Sensor that measures the printing speed.
- P_{ph}: Actuator to set the amount of power supplied to the paper heater.
- P_{rad}: Actuator to set the amount of power supplied to the radiator.

There are two controller modules in the system. The `Paper Heater Controller` controls the paper heater temperature (T_{ph}) to a certain setpoint (T^{sp}_{ph}), by regulating the power to the paper heater (P_{ph}). The setpoint value is configured by other modules, not shown here.

The `Radiator Controller` controls the contact point temperature of the toner belt ($T_{contact}$) to a certain setpoint value ($T^{sp}_{contact}$), by regulating the power to the radiator (P_{rad}).

The control software contains implementations of two physical models. Firstly, it contains the module `Print Quality`

`Model`, which implements a physical model of print quality, based on the temperature and speed variables. The model contains the following physical relationship:

$$T_{contact} = c1 \cdot v - c2 \cdot T_{ph} + c3 \quad (1)$$

In which $c1$, $c2$ and $c3$ are constants. The implementation of this model is used to determine the setpoint value ($T^{sp}_{contact}$) of the `Radiator Controller`, so that print quality is ensured.

Secondly, the control software contains the module `Belt Temperature Model`, which implements a physical model of the belt temperature. This model contains the following physical relationship:

$$T_{contact} = c4 \cdot \frac{P_{rad}}{\sqrt{v}} + T_{belt} \quad (2)$$

This model is used to determine the actual $T_{contact}$, as there is no sensor in the system (due to physical limitations) to directly measure this temperature.

So, the case clearly illustrated a number of physical models implemented in control software. In the next section, we will explain why such models need to be verified.

B. Verifying Physical Relationships

The previous section showed that physical models are being implemented in control software. Such implemented physical models, 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 general-purpose programming language 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.

Incorrectness in an implemented physical model may lead to failures in the behavior of the system. In the embedded software domain it is not possible to test the software in all possible physical conditions. Therefore, certain faults might remain undetected. So, we need to verify physical models.

C. Challenges for the Verification of Physical Models

There are two important challenges regarding the verification of the physical models in embedded software.

1) Failures Observable in Physical System Behavior: Faults in implemented physical models lead to observable failures in the physical behavior of the system, but not necessarily to observable failures in software behavior. This hinders the application of common runtime verification techniques that focus on monitoring software behavior only.

Common runtime verification techniques usually consist of three parts [5]. First, there is a data and/or event model in which the software can be described. Second, there is a logic to specify properties of such models. These are the properties that have to be valid. Examples of such a logic are regular expressions and temporal logics. Third, there is a specification of the actions that need to be performed when the specified properties are violated.

In embedded control software we want to verify whether the assumptions made in the software about the physical reality (i.e., the physical model used in the software) are valid. We want to verify whether the actions performed in the control software lead to the correct behavior of the physical machine. Failures become apparent only in the behavior of the physical system, e.g., in reduced print quality. The same software state might in one case be correct, while under other circumstances it might not be, because it does not necessarily correspond with physical reality. The same reasoning applies to a sequence of events. Common runtime verification techniques are insufficient to do this kind of verification, as their data/event models only include the software state/actions; the state of the physical system is not explicit in the software.

2) Physical Models Implicit in Program Code: The physical models are often expressed in a general-purpose programming language, such as C++, and tangled within control components. This is shown in Listing 1 for the radiator controller. This code fragment shows code that is executed for each iteration of the control loop. The control logic is shown on Line 8. Lines 6 and 7 show the implemented physical model.

```
1  double TcontactSP, Tcontact, Prad;
2
3  /* Control loop */{
4    /* Retrieve values of v, Tph, Tbelt from other
         modules */
5
6    TcontactSP = c1*v - c2*Tph + c3;
7    Tcontact = c4 * Prad / sqrt(v) + Tbelt;
8    Prad = /* classic control logic using tContact
         and tContactSP */;
9
10   /* Send value of pRad to Physical System I/O
         module */
11 }
```

Listing 1. Code fragment of the Radiator Controller

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

III. Solution Approach

In this section we present a novel approach to make physical models explicit and to verify their correctness at runtime.

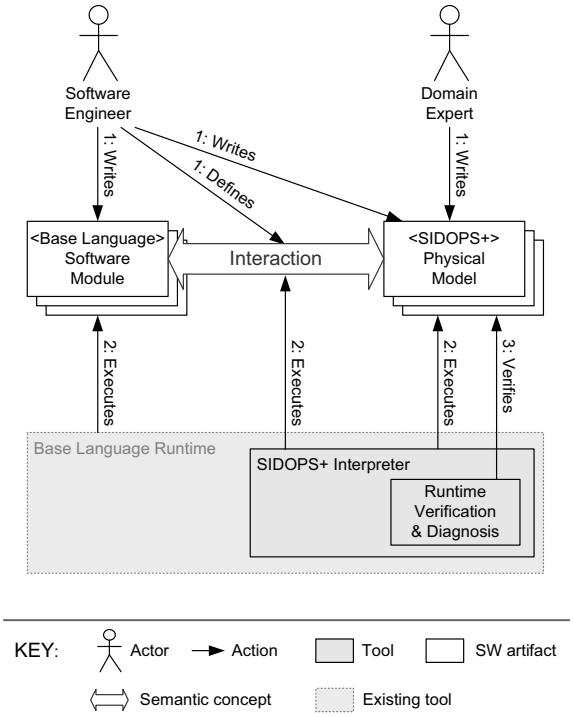

Fig. 3. Overview of our approach

Figure 3 gives an overview of our approach. Instead of implicitly implementing physical models in a general-purpose programming language, in our approach they are separately specified in the domain-specific language SIDOPS+ from the 20-Sim toolset. 20-Sim is a widely adopted toolset with an extensive set of functions generally used for modeling and simulating physical systems [1]. In this section, we will first introduce the SIDOPS+ language. Then we will explain how redundancy is used to verify physical models at runtime. We introduce the concept of dependency graphs to identify redundancy in the physical model and to diagnose detected inconsistencies. This section explains the concepts of our approach. In Section IV we provide details about the realization of the SIDOPS+ interpreter and how monitoring for inconsistencies is performed, using the aspect-oriented composition filters approach.

A. Introduction to 20-Sim/SIDOPS+

The 20-Sim toolset is used for modeling and simulating physical systems. Domain experts can model physical systems in 20-Sim in multiple ways, e.g. using iconic diagrams, bond graphs and the specification language SIDOPS+. In this paper we adopt the SIDOPS+ language. Iconic diagrams and bond graphs can be automatically transformed to SIDOPS+ specifications. As such, a domain expert does not have to use the SIDOPS+ language directly to apply our approach.

With the SIDOPS+ language it is possible to define mathematical physical models, and by using the composition mechanism of the language, they can be integrated to form larger models. Listing 2 gives 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; since physical modeling is not the aim of this paper, they are not explained here.

```
1 constants
2   real c4=2.3;
3 variables
4   real global Tcontact {Temperature, degC};
5   real global Prad=0.0 {Power, W};
6   real global v {PrintSpeed, ppm};
7   real global Tbelt  {Temperature, degC};
8 equations
9   Tcontact = c4 * Prad / sqrt(v) + Tbelt
```

Listing 2. SIDOPS+ example specification

As the name suggests, constants can be defined in the `constants` definition block. The example shows the definition of the constant `c4` 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 `2.3` is given to `c4`.

The `variables` block defines the physical variables. The example shows the definition of four physical variables: `Tcontact`, `Prad`, `v` and `Tbelt`. Variables have a type. They can also have the modifier `global`, which means 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 also shows that a variable can have an initial value assigned. This assignment is optional. It is also possible to attach the name and unit of the corresponding physical quantity to the variable, such as the quantity `Temperature` and unit `degC` for the variable `Tcontact`. 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 also be attached to constant definitions in the same way.

As the name suggests, equations are defined in the `equations` block. Equations are relations between variables. An equation is composed of two mathematical formulas, separated by an equality (=) sign. A mathematical formula can contain variables, constants, operators and predefined functions. The equation in the example shows how the four defined variables relate to each other.

Note that the examples in this paper are simplified for presentational purposes, and as such do not demonstrate the full power of the domain-specific SIDOPS+ language. For example, SIDOPS+ also contains constructs to specify integral

and differential equations, which are commonly used in physical modeling. For further detail about the SIDOPS+ language, we refer to the 20-Sim documentation in [2].

B. Using Redundancy to Verify Correctness

To verify the physical models at runtime, we need to check whether they correspond with actual physical reality. The physical models implemented in software are related to state observers [3] in control engineering. State observers use mathematical physical models to 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 [3]. We will also utilize redundant information about the physical reality, so that the physical relationships can be checked for consistency. This additional information can be:

- *Additional sensors that measure the value of variables that are also calculated with the physical models*. This is a situation that is rare at system deployment, as the physical models are introduced because there is no available sensor information. A purpose may be that the sensor is used for calibration of the model, and only occasionally gives a reading. The industrial case that we will introduce in Section V contains such a sensor.
- *Redundancy in the physical relationships*. If there are multiple physical relationships that calculate the value of the same variable, the results can be compared.

In the warm process case, the `Belt Temperature Model` can also contain the following equation:

$$T_{belt} = c5 \cdot (T_{contact} + c6 \cdot T_{ph}) \cdot \sqrt{v} \tag{3}$$

This equation determines the temperature at the sensor location (T_{belt}) from other physical variables ($T_{contact}$, T_{ph} and v). If this equation is added to the model, there is a redundant way to determine T_{belt}; using the sensor and using the Equation 3. Listing 3 shows the SIDOPS+ specification of the `Belt Temperature Model`.

```
...
equations
  Tcontact = c4 * Prad / sqrt(v) + Tbelt;
  Tbelt = c5 * (Tcontact + c6 * Tph) * sqrt(v);
```

Listing 3. SIDOPS+ specification of the belt temperature model

We can use such redundant ways to calculate/determine the value of a variable to check whether the corresponding physical model is correct. The physical model is correct 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 involved relationships is not correct.

In reality, there will often be a small deviation 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 also be taken into account (i.e. if the difference between the redundant values is within a certain safe range, there is no indication of a failure). In Section IV we will show how we can monitor such inconsistencies, including the filtering of small deviations.

C. Deriving a Dependency Graph

From a specification written in SIDOPS+, a *dependency graph* is created. Such a graph describes how the values of physical variables relate to values of other physical variables, through the different equations specified in the SIDOPS+ model. Figure 4 visualizes a dependency graph for the example in Listing 3. The figure shows, for example, that variable $T_{contact}$ can be derived through Equation 2 from the variables T_{belt}, P_{rad} and v.

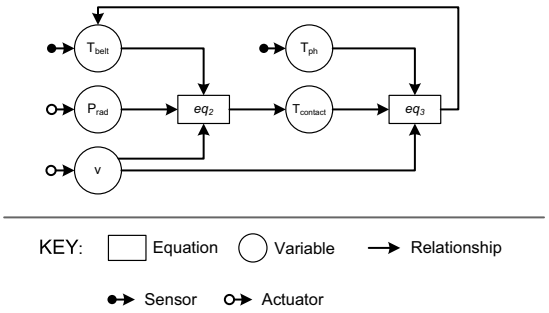

Fig. 4. Dependency graph for the belt temperature model

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. Figure 4 contains one redundant calculation. The variable T_{belt} can be determined in two ways; it can be derived from a sensor input and through Equation 3. Note that there is actually a cycle in the graph, in which T_{belt} depends on itself through Equations 2 and 3. In our example, this means that the new value of T_{belt} is determined from the previous value of T_{belt}.

D. Diagnosing Faults

When the outcomes of redundant calculations are inconsistent, this indicates that there is a failure. The next step is diagnosing the cause of the failure. This can either be a failure of a physical component, e.g. a sensor, or a fault in the implementation of the relationships. Possible causes can be determined from the dependency graph, by tracing back all paths from the variable of which the redundant calculation was not consistent.

For example, suppose that in the printer case the sensor reading for variable T_{belt} gives a different value than the calculation through function $f2$. Using the dependency graph in Figure 4, we can trace back the paths leading to the following possible causes:

1) The sensor value is wrong: this may indicate a malfunctioning sensor.
2) The outcome of Equation 3 is wrong. This may indicate:
   a) Equation 3 itself is incorrect.

b) The input T_{ph} is incorrect. This may indicate a malfunctioning sensor.
c) The input v is incorrect. This may indicate a problem in the actuation/controlling of v, either in software or in hardware.
d) The input T_{pinch} is incorrect. This means that the result of Equation 2 is incorrect, which may indicate:
   i) Equation 2 itself is incorrect.
   ii) The input T_{belt} is incorrect. Because there is a cycle in the dependency graph, by tracing back the paths we end up at the previous value of the variable T_{belt}, which is potentially incorrect. Because of the cycle, the possible causes of this are the recursive closure of this cause list. This closure is finite, because the system was initialized/started a finite time-step in the past.
   iii) The input P_{rad} is incorrect. This indicates a problem in the actuation/controlling of the radiator, either in software or in hardware.
   iv) The input v is incorrect. This case has already been taken into account in point 2c.

IV. Realization of the Approach

This section describes how our approach is realized; how SIDOPS+ models are interpreted and how the aspect-oriented composition filters approach [6] is utilized to monitor physical models for inconsistency, and to handle detected inconsistency.

A. Interpreter

We implemented an interpreter[2] to execute models specified in SIDOPS+. Therefore, we introduced the concept of *physical model instance*, which contains besides the physical relationships also the state of the physical system that is modeled. A physical model instance can be created from one or more SIDOPS+ specifications. A SIDOPS+ specification can be used in multiple physical model instances. Listing 4 shows how a physical model instance can be created, using Java as the general-purpose programming language of the software.

```
1 PhysicalModel beltTemperature =
      PhysicalModel.load("belttemperature.phys");
2 PhysicalModelInstance BeltTemperatureModel = new
      PhysicalModelInstance( new
      PhysicalModel[]{beltTemperature} );
```

Listing 4. Instantiation of the physical model

The interpreter handles the calculations in the model to update the state. It also provides an interface to the physical model instance so that modules in the general-purpose programming language can query the model for values of physical variables and can update the value of certain physical variables in the model. As the focus of this paper is not on how to implement such an interpreter, we omit these details.

[2]We applied an interpreter based approach for research purposes; an interpreter provides the flexibility to quickly implement and change concepts, thus making it easier to experiment with language concepts. For industrial application a compiler based implementation would be preferred.

B. Applying Composition Filters

To monitor for inconsistencies in the physical models, we apply the composition filters approach. The composition filters approach has been applied before as a monitoring approach for runtime verification, such as described in [7]. Figure 5 schematically shows how composition filters are applied to monitor for inconsistencies in the physical model. At runtime, the physical model instance generates different types of events. Some of these events indicate that there are inconsistencies between physical relationships. The composition filters filter these events for relevance and can execute certain behavior based on the event that was captured.

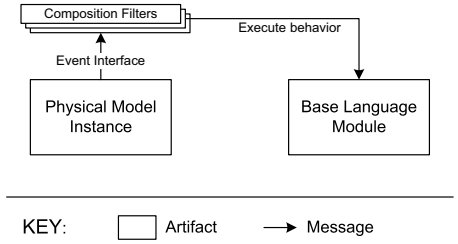

Fig. 5. Monitoring using Composition Filters

If the composition filters capture an event of interest, they can execute certain behavior, either specified by the type of filter or by sending a message to a base-program module. This behavior can for example be logging and reporting the inconsistency or starting a recovery action to recover from known malfunctions.

The event messages that are generated have, among others, the following properties on which they can be filtered:

- *variable*: The identifier of the physical variable on which the event applies.
- *eventType*: The type of the event. Examples of types are:
  - *inconsistency*: The redundant outcomes for the physical variable are not consistent.
  - *update*: The value of the physical variable in the model has changed.
  - *request*: The physical model needs a new value for a certain physical variable, to update the model.
- *inconsistencyRange*: In case the eventType is `inconsistency`, this property indicates how much the different values deviate from each other.
- *value*: This property contains the value of the physical variable. In case the eventType is `update`, this is a single value. In case the eventType is `inconsistency`, this is an array containing all values of the redundant derivations of the physical variable.
- *result*: Property that can be set by composition filter actions, indicating a returned value.

Listing 5 shows a composition filters specification for the warm process case. The purpose of this specification is to log inconsistencies in T_{belt} that are larger than 0.2. Line 3 shows a specification of a `Logging` filter with name `tbeltLog`. A filter of type `Logging` provides logging of

the messages that are matched. The shown filter matches messages for which the property `variable` has value `Tbelt` (matching of events on T_{belt}) and property `eventType` has value `inconsistency` (matching of inconsistency events) and property `inconsistencyRange` has a value larger than 0.2 (matching of inconsistencies that are larger than 0.2).

```
1  filtermodule TbeltLogging{
2    outputfilters
3      tbeltLog: Logging = (variable==Tbelt &
           eventType==inconsistency &
           inconsistencyRange > 0.2);
4  }
5
6  superimposition{
7    selectors
8      models = { M | isModelInstance (M,
           [BeltTemperatureModel]) };
9    filtermodules
10     models <- TbeltLogging;
11 }
```

Listing 5. Composition filters specification to log inconsistencies

Lines 6 till 11 show the superimposition part of the composition filters specification. This part specifies that the filters should be placed on the physical model of the warm process (called `BeltTemperatureModel`).

Listing 6 shows a filter specification that handles the inconsistency, by specifying the value that the physical model should use. The filtertype is `Result`, meaning that if a message is matched by this filter, the filter returns with a given result. The matching part of the filter shows that the filter matches for inconsistency events concerning T_{belt}. After the matching part, this filter has an assignment part (shown on Line 5). In this assignment part, the property `result` is set to the value from the sensor reading (the `value` property is an array, from which a value can be acquired by using an identifier, such as the sensor name in this case).

```
1  filtermodule TbeltHandler{
2    outputfilters
3      tbeltHandler: Result = (variable==Tbelt &
4          eventType==inconsistency)
5          {result=value['Tbelt_sensor']};
6  }
```

Listing 6. Composition filters specification to handle inconsistency

Figure 6 shows both filters in the case's software structure, which was shown in Figure 2.

For further information on the composition filters model, the language and its application in runtime verification in general, we refer to [6]–[9].

V. Application on a Second Industrial Case

This section introduces a second industrial case study, and shows how our approach can be applied to this case study.

A. Drum Shuttling Case Study

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. Figure 7 schematically shows the drum and additional components needed to rotate and shuttle the drum.

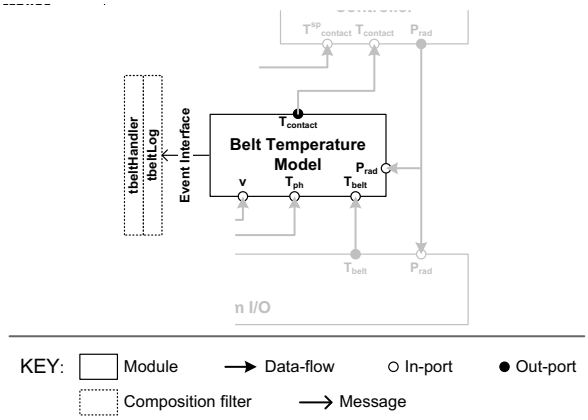

Fig. 6. Specified composition filters applied on Belt Temperature Model

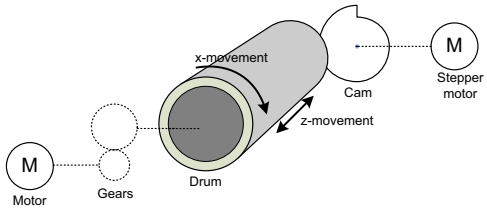

Fig. 7. Schematic view of the drum and components for rotation and shuttling

There is a motor and gears for the rotational movement of the drum. We call this rotation *x-movement*, and the corresponding position (i.e. distance travelled by the surface of the cylinder) *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 *z-position*.

Software has been implemented to control the shuttling behavior of the drum. Figure 8 shows the software structure and the data-flow between the different modules.

The `Physical System I/O` module provides an interface to the following sensors and actuators of the system:

- $pulseCount$: 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.
- dir: Actuator to set the direction in which the stepper motor should step.
- P_{rad}: Actuator that executes one step of the stepper motor when a signal is provided.

The `Shuttling Controller` decides what $zPos$ should be based on the $xPos$. The `Stepper Controller` controls

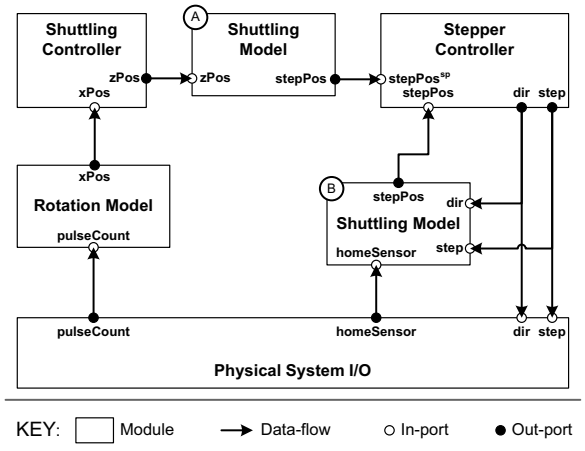

Fig. 8. Schematic overview of the software structure

the stepper motor to a given step position ($stepPos^{sp}$), provided a current step position ($stepPos$) by actuating dir and $step$.

The module `Rotation Model` implements a physical model that models the relationship between $pulseCount$ and $xPos$. It contains physical characteristics of the motor, the gears and the drum.

The module `Shuttling Model` implements a physical model that models the relationship between $zPos$ and $stepPos$, between a signal on $homeSensor$ and $zPos$ and on how a signal on dir and $step$ influence $stepPos$. It thereby contains physical characteristics of the stepper motor, the cam and the drum. Note that this module is actually used two times in the software. The instance labeled `A` is used to translate a requested $zPos$ to the required $stepPos$, which is the setpoint for the `Stepper Controller`. The instance labeled `B` maintains the current step position, based on the output to dir and $step$ and the input from $homeSensor$.

B. Applying the Approach

In the drum shuttling case, the z-position of the drum (z_{pos}) is derived from the step position of the stepper motor. This step position is updated if a step is actuated. However, the stepper motor occasionally does not step when it is actuated, creating a deviation between the physical model and physical reality. Small deviations are no problem, but 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 $homeSensor$ that is triggered when the drum reaches a specific z-position. In this section we will show how this is implemented.

1) Definition of the Physical Model: Listing 7 shows the SIDOPS+ physical model of drum rotation. It shows the translation from $pulseCount$ to $xPos$ through a series of equations.

```
1  // Rotation Model
2  constants
3    real pulsesPerRev=24.0;
4    real gearTransmission=15.0 / 126.0;
5    real drumCircumference=350.0 {Distance, mm};
6  variables
7    integer global pulseCount=0;
8    real global motorRotation {Rotation, rev};
9    real global gearRotation {Rotation, rev};
10   real global drumRotation {Rotation, rev};
11   real global xPos {Distance, mm};
12 equations
13   motorRotation=pulseCount / pulsesPerRev;
14   gearRotation=gearTransmission * motorRotation;
15   drumRotation = gearRotation;
16   xPos = drumRotation * drumCircumference;
```

Listing 7. SIDOPS+ specifications of drum rotation

Listing 8 shows the SIDOPS+ specification of drum shuttling. It shows how $stepPos$ and $zPos$ relate through the equation on Lines 14 and 15. On Line 16 it shows an equation that gives a new $stepPos$, when the stepper motor is actuated (the used function f is not specified further in this paper). Line 17 shows an equation that sets the $zPos$ on the $homeLocation$ if the $homeSensor$ gives a signal (1.0). Otherwise, it maintains the $zPos$ ($homeSensor = 0.0$). If the $zPos$ changes, the interpreter is capable to solve the corresponding $stepPos$, using the equations (the interpreter uses known equation solving algorithms to do this).

```
1  // Shuttling Model
2  constants
3    real degreesPerStep=/* some value */;
4    real movementPerDegree=/* some value */;
5    real homeLocation=/* some value */;
6  variables
7    integer global stepPos=0;
8    real global stepRotation {Rotation, deg};
9    real global zPos {Distance, mm};
10   real global dir;
11   real global step;
12   real global homeSensor;
13 equations
14   stepRotation=stepPos * degreesPerStep;
15   zPos=stepRotation * movementPerDegree;
16   stepPos=f(stepPos, dir, step);
17   zPos=(1.0-homeSensor) * zPos + homeSensor *
         homeLocation;
```

Listing 8. SIDOPS+ specifications of drum shuttling

Listing 9 shows the composition filters specification to monitor and handle inconsistencies in the `Shuttling Model`. Line 3 shows the specification of a composition filter that matches when there is an inconsistency in $zPos$. This is the case if the $homeSensor$ gives a signal, but the $zPos$ derived from the current $stepPos$ in the model is different from the $homeLocation$. The composition filter specifies that in this case the $zPos$ derived using the $homeSensor$ should be used.

```
1  filtermodule sensorHandler{
2    outputfilters
3      sensorHandler: Result = (variable==zPos &
           eventType==inconsistency)
           {result=value['homing_sensor']};
4  }
5
6  filtermodule zPosLogging{
7    outputfilters
8      zPosLog: Logging = (variable==zPos &
           eventType==inconsistency &
           inconsistencyRange > 2);
9  }
```

```
10
11 superimposition{
12   selectors
13     models = { M | isModelInstance (M,
           [ShuttlingModelB]) };
14   filtermodules
15     models <- zPosLogging, sensorHandler;
16 }
```

Listing 9. Composition filters specification to handle home sensor

Line 8 shows a filter that performs logging of the inconsistency if it is larger than two, for later problem diagnosis. Lines 11 until 16 show how the two filter modules are superimposed on the module `ShuttlingModelB`.

VI. Discussion

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

A. Efficiency of the Monitoring Approach

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 works of Bockisch [10].

B. Moment of Checking

The time instance on which a fault/inconsistency in a physical model can lead to a failure is the moment on which that model is evaluated and the result is used by the control modules. Since we want to monitor whether there are inconsistencies in the system that can lead to failures, the best time to do monitoring is after the physical model has been evaluated but before the result is used. In this way, the result of the calculation can also be used in the monitoring (reducing overhead of additional calculations) and we are able to take certain action before the potentially erroneous result of the calculation is used by the control modules, leading to a failure.

One could argue that we can also perform monitoring at other time instances, to diagnose problems in an earlier stage. But this may lead to problems, as the physical relationships might be designed to only be consistent at the time they are evaluated. Furthermore, this reduces performance.

C. Recovery Actions

If the different redundant values of a certain physical variables do not match, a recovery action needs to 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.

D. Integration with System Architecting Models

In this paper we used the domain-specific SIDOPS+ language to make the physical models used in embedded software explicit. In this way, domain engineers are able to read the specification, check for correctness and are even able to write the specifications themselves. By using domain-specific languages and models, it becomes possible to integrate them with other system architecting models. System architects use tools to model different aspects of the physical system, such as the physical structure, physical behavior and interaction between components in the system, state of the system etc. As the models of physical characteristics are heavily based on such system models, integration is a possibility.

VII. Related Work

Literature shows, e.g. in the taxonomy of Delgado et al. [5] and in the work of Barringer et al. [11], that the common approach to runtime verification is to create a data and/or event model in which the software can be described and to verify certain properties on this model, specified in a certain logic, such as temporal logic or regular expressions. Examples of such runtime verification approaches are the MOP framework [12] and the tracematches extension to AspectJ [13]. Such approaches cannot be applied in our case, as we need to take the impact of the software behavior on the physical behavior of the system (i.e. the operating environment of the software) into account; failures only become apparent in the physical behavior. Therefore, our approach uses redundant models of physical relationships to verify their conformance with physical reality.

Van Gemund et al. have worked on fault diagnosis in embedded systems [14]. 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 [15] and De Kleer [16], 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 [14]. Van Gemund et al. combined both approaches to be applied on the combination of embedded system and the corresponding embedded software [17], [18].

Work is being done on integrating system architecting models, describing the system from different perspective, so that consistency can be maintained and changes in one model can be automatically reflected in other models. An example of such an effort is the Knowledge Intensive Engineering Framework (KIEF), described in [19]. The model of physical characteristics and the dependency graphs we use are closely related to models used in these system architecting tools. For example, Forbus describes in [20] the concept of qualitative process theory. This is the analysis and specification of the qualitative

relationships between different physical quantities in a physical system. Such a specification can be used to predict behavior in the system. This theory has been incorporated in KIEF in the form of parameter networks. Parameter networks are graph structures that describe the qualitative relationships between physical parameters [19]. They can be derived from other system models, like structural models. Parameter networks are similar to the dependency graphs presented in this paper. The difference between them is that dependency graphs provide a quantification on these relationships, while parameter networks only qualitatively describes the relationships. So, parameter networks reflect the graph structure of dependency graphs, but without the function nodes in the graph. As such, the parameter networks can be used as a starting point for a SIDOPS+ specification.

VIII. Conclusion and Future Work

In this paper we showed that models of physical characteristics (i.e., physical models) are part of control software. Such models are used for estimating physical relationships among system components, and influencing the control behavior accordingly. As such, faults that are undetected in physical models can lead to a failure in control behavior. Therefore, it is important to monitor and verify the accuracy of these models at runtime.

However, traditional runtime verification techniques cannot be applied, as their focus is on verifying whether software behavior corresponds to a separately specified model of correct behavior. Our aim is to verify physical models used in software for their correspondence with physical reality. Furthermore, implicit implementation of physical models in a general-purpose programming language makes these models hard to locate and verify.

We showed that it is possible to specify the physical models used in software in the domain-specific SIDOPS+ language from the 20-Sim toolset. Such specifications are applied as part of the software, using an interpreter based approach. We showed that redundancy can be identified in such specifications, using dependency graphs. The redundancy can be exploited to detect inconsistencies in the physical model or malfunctions of related sensors and actuators in the system. The dependency graph can be further utilized for diagnosing root causes of failures (i.e., faults) in physical models or malfunctioning sensor/actuator components in the system, by tracing back the paths from an inconsistent variable node. We showed that monitors can be generated using the aspect-oriented composition filters language.

Future work is the integration of our tools with system architecting tools, to better align the different disciplines of embedded systems development.

Furthermore, we want to analyze how different levels of redundancy influence the accuracy of the diagnosis. In this paper we have shown that diagnosis can be done by tracing back the paths from the failing node in the dependency graph. If there are more redundant nodes, such paths may become shorter, making the analysis more precise.

Acknowledgment

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 this paper and providing useful feedback.

References

[1] J. Broenink, "Modelling, simulation and analysis with 20-sim," *Journal A*, vol. 38, no. 3, pp. 22–25, 1997.

[2] C. Kleijn, *20-sim 4.1 Reference Manual*, 2009.

[3] E. D. Sontag, *Mathematical Control Theory: Deterministic Finite Dimensional Systems*, 2nd ed. New York: Springer, 1998.

[4] "Octopus project, ESI," 2010, http://www.esi.nl/projects/octopus.

[5] N. Delgado, A. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *Software Engineering, IEEE Transactions on*, vol. 30, no. 12, pp. 859 – 872, dec. 2004.

[6] A. J. de Roo, M. F. H. Hendriks, W. K. Havinga, P. E. A. Durr, and L. M. J. Bergmans, "Compose*: a language- and platform-independent aspect compiler for composition filters," in *First International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008, Paphos, Cyprus*, July 2008.

[7] S. Malakuti Khah Olun Abadi, C. M. Bockisch, and M. Akşit, "Applying the composition filter model for runtime verification of multiple-language software," in *The 20th annual International Symposium on Software Reliability Engineering, ISSRE 2009, Mysore, India*. IEEE Computer Society Press, 2009, pp. 31–40.

[8] University of Twente, "COMPOSE*," http://composestar.sourceforge.net.

[9] *Compose* Annotated Reference Manual*, Internet: http://composestar.svn.sourceforge.net/viewvc/composestar/documentation/ARM/ARM.pdf, University of Twente.

[10] C.-M. Bockisch, "An efficient and flexible implementation of aspect-oriented languages," Ph.D. dissertation, Technische Universität Darmstadt, Germany, July 2008.

[11] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *VMCAI*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds., vol. 2937. Springer, 2004, pp. 44–57.

[12] F. Chen and G. Roşu, "Mop: an efficient and generic runtime verification framework," in *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. New York, NY, USA: ACM, 2007, pp. 569–588.

[13] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 345–364.

[14] P. Zoeteweij, J. Pietersma, R. Abreu, A. Feldman, and A. van Gemund, "Automated fault diagnosis in embedded systems," in *Secure System Integration and Reliability Improvement, 2008. SSIRI '08. Second International Conference on*, 14-17 2008, pp. 103 –110.

[15] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[16] J. de Kleer and B. C. Williams, "Diagnosing multiple faults," *Artificial Intelligence*, vol. 32, no. 1, pp. 97–130, 1987.

[17] R. Abreu, P. Zoeteweij, and A. van Gemund, "Spectrum-based multiple fault localization," in *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, 16-20 2009, pp. 88 –99.

[18] A. Feldman, G. Provan, and A. van Gemund, "The Lydia approach to combinational model-based diagnosis," in *Proceedings of the Twentieth International Workshop on Principles of Diagnosis (DX'09), Stockholm Sweden*. Erik Frisk and Mattias Nyberg and Mattias Krysander and Jan Åslund, June 2009, pp. 403–408.

[19] M. Yoshioka, Y. Umeda, H. Takeda, Y. Shimomura, Y. Nomaguchi, and T. Tomiyama, "Physical concept ontology for the knowledge intensive engineering framework," *Advanced Engineering Informatics*, vol. 18, no. 2, pp. 95 – 113, 2004.

[20] K. D. Forbus, "Qualitative process theory," *Artificial Intelligence*, vol. 24, no. 1-3, pp. 85 – 168, 1984.