Softw Syst Model
DOI 10.1007/s10270-012-0283-7

THEME SECTION PAPER

Composing domain-specific physical models with general-purpose software modules in embedded control software

Arjan de Roo · Hasan Sözer · Mehmet Akşit

Received: 1 December 2010 / Revised: 26 April 2012 / Accepted: 20 August 2012

Abstract A considerable portion of 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. In current practices, usually general-purpose languages (GPL), such as C/C++ are used for embedded systems development. Although a GPL is suitable for expressing general-purpose computation, it falls short in expressing the models of physical characteristics as desired. This reduces not only the readability of the code but also hampers reuse due to the lack of dedicated abstractions and composition operators. Moreover, domain-specific static and dynamic checks may not be applied effectively. There exist domain-specific modeling languages (DSML) and tools to specify models of physical characteristics. Although they are commonly used for simulation and documentation of physical systems, they are often not used to implement embedded control software. This is due to the fact that these DSMLs are not suitable to express the general-purpose computation and they cannot be easily composed with other software modules that are implemented in GPL. This paper presents a novel approach to combine a DSML to model physical characteristics and a GPL to implement general-purpose computation. The composition filters model is used to compose models specified in the DSML with modules specified in the GPL at the abstraction level of both languages. As such, this approach combines the benefits of using a DSML to model physical characteristics with the freedom of a GPL to implement general-purpose computation. The approach is illustrated using two industrial case studies from the printing systems domain.

Keywords Domain specific languages · Embedded systems · Software composition · Composition filters · Aspect-oriented programming

Communicated by Dr. Jeff Gray, Juha-Pekka Tolvanen, and Matti Rossi.

A. de Roo (✉) · M. Akşit
Software Engineering group, CS Department,
University of Twente, Enschede, The Netherlands
e-mail: roo@ewi.utwente.nl

H. Sözer
Computer Science Department,
Özyeğin University, İstanbul, Turkey

1 Introduction

A considerable portion of software systems today are adopted in the embedded domain (e.g., medical equipment, military applications, traffic control systems, consumer electronics) [54]. A major portion of embedded systems aim at controlling physical systems in some way and as such, the characteristics of the physical systems must be represented in Software accordingly. For example, in state-of-the-art printing systems, the speed of the machine is adapted according to the available power, temperature measurements obtained from several components and the heat capacity. The embedded software has to consider such physical characteristics to apply a particular control strategy.

In embedded systems development, the current practice is to use general-purpose programming languages (GPLs) such as C and C++. Usually, both the control logic (i.e., computation of the controlling behavior) and the models of physical characteristics are implemented together in a GPL. The control logic and the physical characteristics are two different concerns and the lack of separation of these concerns leads to implicit and complex dependencies in the code due to tangling and scattering [11]. This results in decreased software readability and maintainability. Changes in hardware

Published online: 07 October 2012

are common for continuously evolving embedded systems. These changes may cause ripple effects within the implementation of modules, if the separation of physical concerns and controlling concerns are not realized properly. This is also confirmed by several publications in modeling [6,26,32], where it seems natural to separate the code that is responsible for interactions, from the code that implements the core behavior of modules. In embedded software, implemented models of physical characteristics are responsible for additional interaction among the GPL modules that implement control logic.

One can utilize object-oriented (OO) design patterns [27] and/or aspect-oriented software development (AOSD) approaches [24] to modularize the code better, and as such facilitate the separation of concerns. However, even if modularized, the models of physical characteristics would still be implemented in a GPL. GPLs have insufficient expression power to define physical characteristics. Moreover, implicit implementation and the lack of domain-specific abstractions hinders the possibility for effective static/dynamic domain-specific analysis that can be applied to ensure their reliability.

Instead of implementing embedded control software in a GPL, one might suggest to use a domain-specific modeling language (DSML), such as Matlab Simulink [13] or 20-Sim [1,16]. These languages are very suitable to express models of physical characteristics and continuous control logic. However, control software for embedded systems usually also implements other application logic, such as logic to schedule (discrete) tasks, to recover from errors, and to monitor the available resources (e.g., the amount of toner, number of sheets of paper in the paper tray). DSMLs to model physical characteristics are not particularly suitable to express these types of functionality. Therefore, this functionality is usually, and more effectively, implemented in a GPL.

DSMLs, such as Matlab Simulink and 20-Sim, offer to possibility to generate GPL code modules from DSML models. However, these generated code modules often have limited interfaces; other software modules have to be fitted around them, leading to tightly coupled software modules. Furthermore, the generated code is not intended to be human readable and as such becomes a black box in software. Therefore, often it is decided to implement models of physical characteristics and continuous control logic in a GPL instead of using a DSML combined with code generation.

Hence, we need to be able to develop artifacts in both GPL and DSML separately, and we need to be able to combine these artifacts. In this paper, we propose a method to compose physical models specified in a DSML with software modules specified in a GPL at the abstraction level of both languages. We adopt the domain-specific modeling language (DSML) *SIDOPS+* of the *20-Sim* toolset [1] to express the logic that deals with physical characteristics of the system. To compose physical models specified in SIDOPS+ with other software modules specified in a GPL, we apply the existing Composition Filters model, which is implemented in the Compose* language and toolset [48]. The Composition Filters modelenables loose coupling between software modules and physical models. Furthermore, we extended the Composition Filters modelto enable interaction using messages and events for GPL modules and DSML models, respectively. To facilitate this, we defined an event model on the execution semantics of DSML models. As such, our approach combines the benefits (e.g., ease of realization, maintainability, reusability) of a DSML to implement models of physical characteristics, with the freedom of a GPL to implement other application logic.

Figure 1 shows how the different concepts in our approach are related to each other. There are three types of development artifacts: application logic implemented in GPL modules, physical models specified in a DSML and composition filters specified in the Compose* language. The interaction between software modules and physical models is represented by messages and events. This interaction is enabled by a number of composition filters; these composition filters specify which events in the execution of the physical model are interesting (e.g., the change of the value of a physical variable) and they specify how these interesting events are processed (for example, the event is dispatched to a certain software module or the event is logged). We have defined an event model for physical models. This event model specifies which events in the execution of a physical model can be selected/quantified and what properties these events have. The composition filters can use these properties to select events of interest. The GPL modules are executed by a GPL runtime environment. An interpreter has been implemented to execute the physical models and the composition filter specifications.

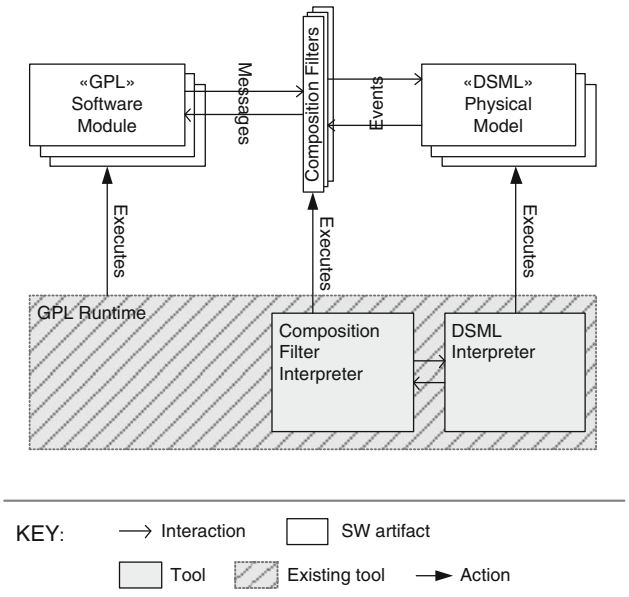

Fig. 1 Overview of our approach

Specification of physical characteristics with a DSML enables static analysis for detecting faults at design time. However, it is not possible to test embedded software in all possible physical conditions and detect all potential faults. Therefore, we complement static analysis with dynamic analysis. Runtime monitors are automatically generated for runtime verification of the specified physical characteristics.

The remainder of this paper is organized as follows. In the next section, we introduce two industrial case studies, taken from the domain of digital document printing systems, to illustrate the problem and our approach. In Sect. 3, we provide the problem statement. An overview of our approach is presented in Sect. 4. Sections 5 and 6 describe, respectively, how physical models are executed and how they are composed with GPL modules. Section 7 gives an overview on domain-specific analysis techniques and explains how the consistency of the physical models can be verified at runtime. In Sect. 8 we apply our approach to the two case studies and evaluate the benefits using evolution scenarios. Section 9 discusses some additional aspects of our approach. Related work is provided in Sect. 10. Finally, in Sect. 11 we discuss future work and provide the conclusions.

2 Industrial cases: digital document printing systems

In this section, we introduce two industrial case studies, taken from the digital document printing systems domain. These case studies have been developed and evaluated within the context of the Octopus project [42], where Océ-Technologies B.V. (one of the world's leading manufacturers of printer and copier systems) is the carrying industrial partner. Although the case studies are simplified for presentational purposes, they are still relevant for illustrating the problem and solution approach discussed in this paper. Each case describes (1) an overview of the controlled hardware, (2) the related control components and physical characteristics that have to be implemented, and (3) the way that the physical characteristics are currently implemented in control components.

2.1 Case I: warm process

The first case that we are going to introduce is called the *warm process*, which is responsible for transferring a *toner image* to paper.

2.1.1 Hardware

Figure 2 gives a schematic view of the components in the printing system responsible for the warm process behavior. The warm process has two main parts; a *paper path* for transporting a sheet of paper and a *toner belt* for transporting a toner image. For correct printing, both the sheets of paper as well as the toner belt should have a certain temperature

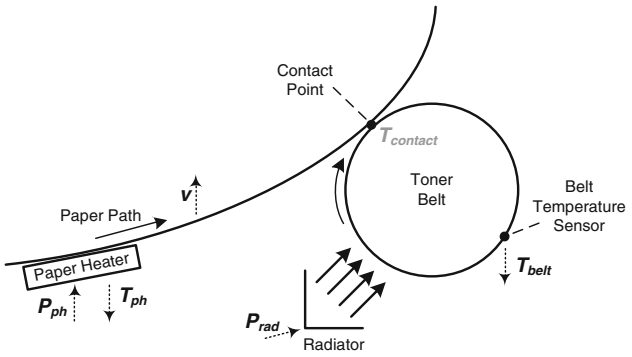

Fig. 2 Schematic view of the *warm process*

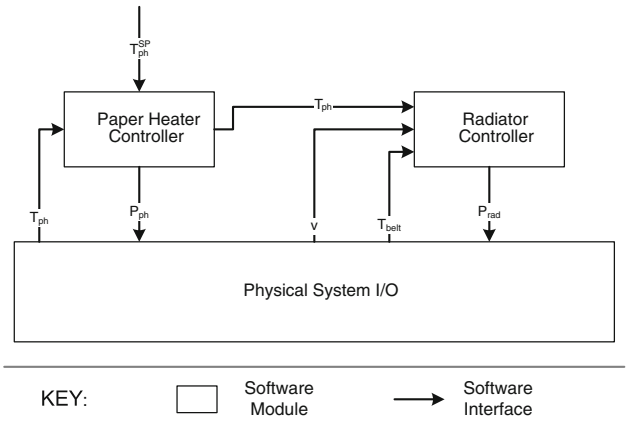

Fig. 3 Schematic overview of the software structure

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.

2.1.2 Control software

Software has been implemented to control the heaters in order to maintain required temperatures for correct printing. Figure 3 shows the related GPL modules.

The *Physical System I/O* module provides an interface to the sensors and actuators of the system. The sensors and actuators are:

- Sensors:
  - T_{ph} The temperature of the paper heater.
  - T_{belt} The temperature at the sensor location on the toner belt.
  - v: The printing speed.
- Actuators:
  - P_{ph} The amount of power supplied to the paper heater.
  - P_{rad} The amount of power supplied to the radiator.

There are two modules implementing control logic in the system. The `Paper Heater Controller` adjusts the

paper heater temperature (T_{ph}) to a certain setpoint ($T_{\text{ph}}^{\text{SP}}$), by regulating the power to the paper heater (P_{ph}). The setpoint value is configured by other modules, not shown here.

The `Radiator Controller` adjusts the contact point temperature of the toner belt (T_{contact}) to a certain setpoint value ($T_{\text{contact}}^{\text{SP}}$), by regulating the power to the radiator (P_{rad}). The setpoint value is determined from the paper heater temperature and the speed, using the following physical relationship, which ensures correct print quality:

$$T_{\text{contact}}^{\text{SP}} = c1 \cdot v - c2 \cdot T_{\text{ph}} + c3 \tag{1}$$

In which $c1$, $c2$ and $c3$ are constants.

Due to physical limitations, there is no sensor to measure T_{contact}, and as such the `Radiator Controller` also implements the following physical relationship that derives this temperature from T_{belt}, v and P_{rad}:

$$T_{\text{contact}} = c4 \frac{P_{\text{rad}}}{\sqrt{v}} + T_{\text{belt}} \tag{2}$$

Typically, such physical relationships are implemented in the code of the `Radiator Controller` module as illustrated in the following listing.

```
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

The listing shows that the physical characteristics (lines 6 and 7) are tangled with the control logic (line 8). This reduces maintainability and reusability.

2.2 Case 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 has to shuttle (i.e., move backward and forward) along its axis.

2.2.1 Hardware

Figure 4 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*.

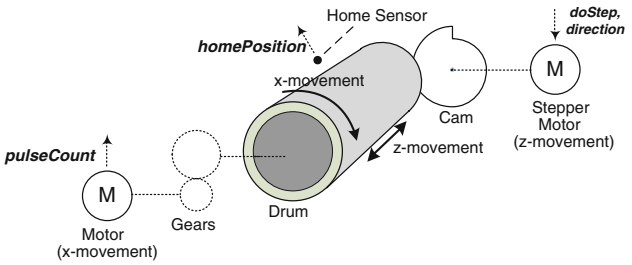

Fig. 4 Schematic view of the Drum

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.
- $homePosition$ Sensor that gives a signal when the drum is at the home position.
- $direction$ Actuator to set the direction in which the stepper motor should step.
- $doStep$ Actuator that executes one step of the stepper motor when a signal is provided.

2.2.2 Control software

A controller component decides what the z-position should be based on the x-position.

For economical reasons there is no sensor to directly measure the x-position. Instead, the pulses that the drum motor gives at a fixed rate per revolution are counted. Then the x-position ($xPos$) is calculated using the following equation:

$$xPos = \frac{pulseCount}{C_{pulsesPerRev}} \cdot C_{gearRatio} \cdot C_{drumCircumference} \tag{3}$$

In this equation, $pulseCount$ is the sensor reading. There are three constants:

- $C_{pulsesPerRev}$ represents the amount of hallpulses the motor gives per revolution.
- $C_{gearRatio}$ this is the relative relationship between the number of teeth on the gears and as such represents the

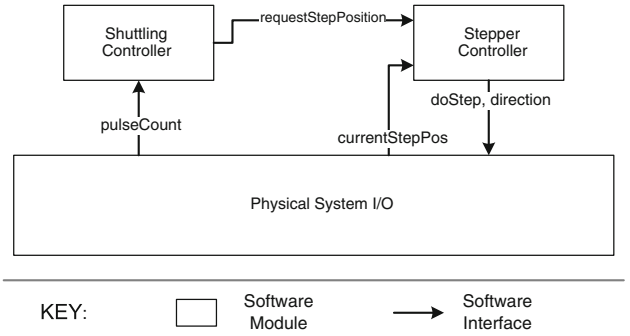

Fig. 5 Schematic overview of the software structure

number of revolutions the drum makes when the motor makes one revolution.
- $C_{drumCircumference}$ the circumference of the drum.

The z-position ($zPos$) of the drum can only be controlled by actuating the stepper motor to a given step-position ($stepPos$). The step-position is derived from the z-position using the following equation:

$$stepPos = \frac{zPos}{C_{degreesPerStep} \cdot C_{zMovementPerDegree}} \quad (4)$$

This equation has the following two constants:

- $C_{degreesPerStep}$ represents the number of degrees the stepper motor turns with each step, and thus the number of degrees the cam turns with each step.
- $C_{zMovementPerDegree}$ this is the amount of zMovement of the drum for each degree the cam turns. The used cam provides z-movement linear in the rotation.

Figure 5 shows schematically the related modules of the control software. The module `Shuttling Controller` controls the shuttling behavior. Listing 2 gives a code fragment of this module, showing the implementation of Eqs. 3 and 4 on lines 6 and 8, respectively. The module `Stepper Controller` controls the stepper motor to a certain given step-position, using the current step-position as input from a sensor and some actuators to make steps.

```
1  double xPos, zPos
2
3  /* Control loop */{
4    /* Retrieve value of pulseCount from
         Physical System I/O module */
5
6    xPos = pulseCount / CpulsesPerRev *
         CgearTransmission *
         CdrumCircumference;
7    zPos = /* decision logic using xPos */;
8    stepPos = zPos / CdegreesPerStep /
         CzMovementPerDegree;
9
10   /* Send value of stepPos to Stepper
         Controller module */
11 }
```

Listing 2 Code fragment of the Shuttling Behavior module

Similar to the first case, the equations that represent the physical characteristics are tangled with the control logic (line 7), reducing maintainability and reusability. The next section explores these problems in depth.

3 Problem statement

The case studies introduced in the previous section show that models of physical characteristics (physical models) are part of the control logic implemented in embedded control software. Especially when systems are designed to be adaptive, more physical models are implemented in embedded control software. However, such implemented physical models introduce additional complexity in software. If this complexity is not managed properly, it will reduce software quality. This section analyzes the issues related to physical models that are implemented in embedded control software. First, the section briefly explains the current state-of-the-practice approaches. Then, it compares applying a general-purpose programming language (GPL) versus applying a domain-specific modeling language (DSML) to implement physical models. Benefits and drawbacks of both approaches are analyzed. Finally, this section summarizes characteristics of an approach that addresses the observed issues.

3.1 Current state-of-the-practice

3.1.1 Continuous evolution

During the cooperation with our industrial partner, we have identified that the design of an embedded system evolves continuously due to changing customer needs, technological advances and cost reductions. Examples of evolution scenarios, in the context of the industrial case studies introduced in Sect. 2, are replacing the heaters with different types (having other properties) or changing the gears between the motor and the drum. These changes to the design of the embedded system affect the physical characteristics of the system. When such changes have to be realized, software engineers have to locate the corresponding pieces of code that are affected by these changes and modify them according to the new design.

3.1.2 Design process

In current practice, two main steps are commonly adopted in the development of embedded systems. First, physical behavior is modeled and simulated using tools like 20-Sim [1,16] and Matlab Simulink [13]. The purpose of this step is to analyze the behavior of the embedded control system through simulations. Second, an implementation of the embedded control software is realized based on the analysis results. For this realization, two techniques are used: Traditional programming techniques using GPLs like C and C++, and

model-driven engineering techniques based on DSMLs for physical models and code generation from DSML models to implementations in a GPL. In the following subsections, we discuss issues and limitations regarding these alternatives and the combination of the two.

3.2 Development with GPLs

General-purpose programming languages lack abstractions specific for physical models (i.e., domain-specific abstractions). Therefore, when physical models are implemented in embedded control software, their domain-specific abstractions are translated to general-purpose abstractions in the GPL. The loss of domain-specific abstractions makes it harder to recognize physical models in embedded control software as such. We call this *implicit implementation* of the physical characteristics. During the cooperation with our industrial partner, we have indeed identified that this leads to the following issues.

3.2.1 Physical characteristics harder to locate

Implicit implementation makes it hard to locate the implementation of a physical characteristics in case a change is necessary due to evolution. This in turn, increases maintenance costs and might reduce reliability (when not all relevant code is updated consistently).

3.2.2 Tangling and scattering of physical models

Because of implicit implementation, there are no clear boundaries between physical models and other application logic. This may lead to tangling and scattering of physical models in embedded control software. Listings 1 and 2 of the industrial case studies show tangling of physical models with control logic in GPL module code. As frequently claimed in the aspect-oriented programming literature [24], tangling and scattering of *concerns* can lead to higher maintenance costs through reduced comprehensibility and evolvability of the tangled concerns.

3.2.3 Introduction of accidental complexity

Translation of domain-specific abstractions to implementation abstractions might introduce accidental complexity [17]. Accidental complexity reduces comprehensibility and maintainability of software systems.

3.2.4 Reduced domain-specific analysis

Implicit implementation hinders domain-specific analysis of the implemented physical characteristics for detection of faults, as domain-specific abstractions are lost. This makes it harder to ensure the reliability of the control software. Incorrect or inaccurate implementation of physical characteristics and the lack of domain-specific analysis pose a threat to reliability. The implementation of physical characteristics may not always be accurate or correct due to several reasons, e.g.:

1. The engineer implementing a physical characteristic may introduce a fault;
2. Physical characteristics may change because of evolution (e.g., a change in hardware). Implicit and tangled implementation of characteristics increases the probability that engineers loose track of them in the code; when the physical system is changed, there is a higher probability that the corresponding physical characteristics in the software are not updated, introducing faults in the control software. Moreover, while locating and correcting the affected tangled modules, new faults can be introduced;
3. The underlying assumptions on which the physical characteristics are based may not be accurate enough. For example, the physical characteristics may not accurately describe the physical reality.

The resulting errors may remain undetected due to complex mathematical dependencies and lack of domain-specific analysis.

3.3 Development based on DSMLs

Domain-specific modeling languages reduce or eliminate accidental complexity that is introduced by translating domain-specific abstractions to abstractions in the implementation language. This improves the maintainability of software, as has been proven in several publications such as [20,34]. For instance, Matlab Simulink [13] and 20-Sim [1] provide DSMLs that are suitable to model physical characteristics, as well as continuous control logic. The accompanying tooling offers the possibility to compile models into executable software [1,13]. The tooling can also apply domain-specific analysis to detect faults prior to code generation.

However, control software for embedded systems usually contains other application-specific functionality that does not fall in the categories of physical models and continuous control logic. Examples of such functionality are managing system states (e.g., idle, start-up, available), scheduling of (discrete) tasks, recovering from errors, monitoring the available resources (e.g., the amount of toner, number of sheets of paper in the paper tray), processing of user input, security, communication, maintenance tasks, interaction with third-party libraries, etc. It is impractical to design or adopt a DSML to express these types of functionality, which are commonly expressed in a GPL. Therefore, DSML models should interact with GPL modules.

Models that are expressed in DSMLs can be compiled into GPL modules and their interaction/composition with other GPL modules are usually realized in terms of pre-designed function calls. For example, through *S-functions* Simulink models can call GPL code [13]. These constructs put specific constraints on the interaction; S-functions assume a particular structure of the GPL function; generated GPL modules are translations of the models and provide a specific interface. These constraints create tight dependencies between the GPL modules and DSML models involved in the interaction. As a result, GPL software modules generated from domain-specific models become black boxes within the software architecture. Software engineers cannot easily understand and maintain them. Moreover, the software architecture gets constrained by the generated code: the other GPL modules have to be fitted around the generated modules, work-arounds need to be implemented if not all required interactions with the physical model are possible with the generated code, etc. Because of these limitations, in current practice the physical models are usually implemented and maintained in the GPL. DSML models are usually created by domain engineers for documentation and simulation purposes only.

To prevent the aforementioned problems and to be able to exploit the created domain models in software, the composition of DSML models with GPL modules on the abstraction level of both languages should be facilitated.

3.4 Composing DSML and GPL artifacts

Based on the observations noted in the previous section, it can be concluded that DSMLs are the preferred method to specify physical models. However, DSMLs are not used to implement physical models in embedded control software because of the limited ability to compose physical models specified in a DSML with other software modules specified in a GPL and the strict and tightly coupled interfaces resulting from existing code generation techniques.

An improvement of the current situation would be possible with an approach that offers flexible composition between physical models specified in a DSML and software modules written in a GPL, on the abstraction level of both the DSML and the GPL. Such a method combines the benefits of a DSML to implement physical models with the freedom of a GPL to implement other application logic. Furthermore, using a DSML provides separation of physical models from other application logic. Together with flexible composition operators, this separation prevents tangling and scattering of physical models with/through other application logic. Composition at the abstraction level of both languages eliminates the need to first translate DSML models to GPL modules (i.e., code generation), before the physical models can be composed with other software modules. This prevents tight integration caused by the limitations of generated interfaces.

3.5 Ensuring the reliability at runtime

In the embedded software domain it is not possible to test the software in all possible physical conditions. Even if static analysis are applied, some faults may remain undetected due to several reasons, e.g.,

1. 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.;
2. Physical characteristics may change over time, because of, for instance, wear and tear of physical components in the system.

As such, it is also necessary to verify the implementation of physical characteristics at runtime. Common runtime verification techniques [19] 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 implicit in the software.

Explicit specification of physical characteristics in DSMLs can be utilized for generating monitoring code that checks their consistency at runtime. However, the models and their interaction become obscure/implicit in the generated and composed software. Hence, runtime analysis and interpretation of these models are limited. The applied analysis can only be based on the instrumentation of the code, which is constrained by the application-specific composition.

4 Approach overview

Figure 6 schematically shows our approach to deal with the issues introduced in the previous section.

The figure shows three different types of artifacts: software modules written in a GPL, physical models specified in a DSML and composition 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 applied toolset with an extensive set of functions suitable for modeling and simulating physical systems [1,16]. In the following we call a SIDOPS+ specifi-

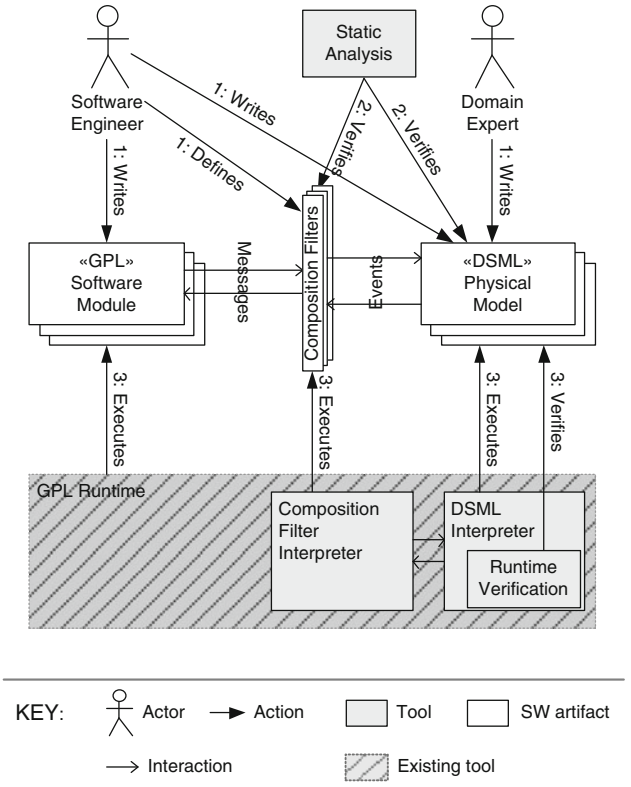

Fig. 6 Detailed overview of our approach

cation a *physical model*, or just *model*. We defined execution semantics for physical models in embedded control software[1] and implemented an interpreter to execute the physical models. Section 5 explains in detail how physical models are specified and executed.

To compose physical models with software modules, we apply the Composition Filters model. To facilitate the application of the Composition Filters model, an event model has been defined on top of the execution semantics of physical models. This event model defines several types of events that occur during the execution of a physical model, and it defines the properties of these events. Specified *composition filters* filter events and execute certain behavior when an event matches. This behavior is for example the dispatch of a certain message to a GPL software module or logging the occurrence of certain events. The dispatch functionality of the Composition Filters modelprovides a transformation mechanism between events and messages.

Applying a DSML to specify physical models provides the possibility to do domain-specific analysis, which obviously is not provided in mainstream languages like C or Java. Section 7.1 summarizes a number of existing domain-specific static analysis techniques. Certain issues cannot be detected statically; Sect. 7.2 explains how runtime verification of physical models can be performed. This verification is facilitated by the DSML interpreter and the Composition Filters model.

[1] There are some small differences with the execution semantics of physical models in the 20-Sim simulation tooling. These differences are discussed in Sect. 9.5.

5 Specifying and executing models of physical characteristics

This section explains how physical models are represented in embedded control software using the SIDOPS+ language from the 20-Sim toolset and how physical models are executed in embedded control software.

5.1 Introduction to 20-Sim/SIDOPS+

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 [37].

Listing 3 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.

```
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;
```

Listing 3 20-Sim example specification

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 indi-

cates 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.

For further detail about the SIDOPS+ language, we refer to the 20-Sim documentation in [37].

5.2 Composing 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.

5.2.1 Example: composition of physical models

Listings 4, 5 and 6 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.

```
constants
  real pulsesPerRev=24.0;
variables
  integer global pulseCount=0;
  real global motorRotation {Rotation,
      rev};
equations
  motorRotation=pulseCount / pulsesPerRev;
       // eq1
```

Listing 4 SIDOPS+ model of motor rotation (`motor`)

```
constants
  real gearTransmission=15.0 / 126.0;
variables
  real global motorRotation {Rotation,
      rev};
  real global gearRotation {Rotation,
      rev};
equations
  gearRotation=gearTransmission *
      motorRotation; // eq2
```

Listing 5 SIDOPS+ model of gear rotation (`gears`)

```
constants
  real drumCircumference=350.0 {Distance,
      mm};
variables
  real global gearRotation {Rotation,
      rev};
  real global drumRotation {Rotation,
      rev};
  real global xPosition {Distance, mm};
equations
  drumRotation = gearRotation; // eq3
  xPosition = drumRotation *
      drumCircumference; // eq4
```

Listing 6 SIDOPS+ model of drum rotation (`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`. Figure 7 shows these interaction points schematically, using graphs to represent the dependencies between the variables and the equations.

5.3 Instantiation of physical models

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.

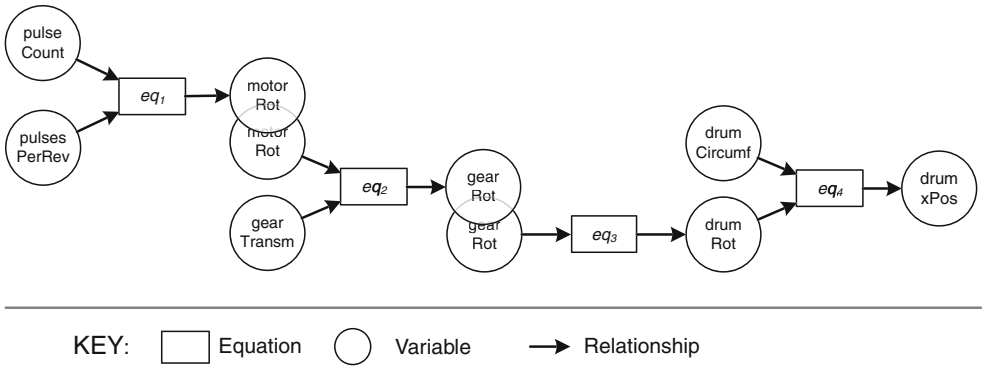

Fig. 7 Interaction points between models

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 Figure 8 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 studythis 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 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 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 $a = 2 * b$. 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 $2 * b$.
  Such an application of physical model instances can be used in the Warm Process case study, to determine the required value T_{contact} (i.e., $T^{\text{sp}}_{\text{contact}}$), based on the current values of T_{ph} and v (Eq. 1).

5.4 Executing physical models

At runtime, the values in the physical state of a physical model instance can be changed, for example based on sensor readings (Sect. 6 describes the interface to the physical model instance, which enables software modules to change values of physical variables). When the values of certain variables in the physical state change, the physical model instance ensures that the physical state remains consistent with the corresponding physical model. This means that the physical model instance executes the physical model to determine a new value of each physical variable, using the equations in the physical model and the values of the physical variables that have been changed. To perform this operation, the phys-

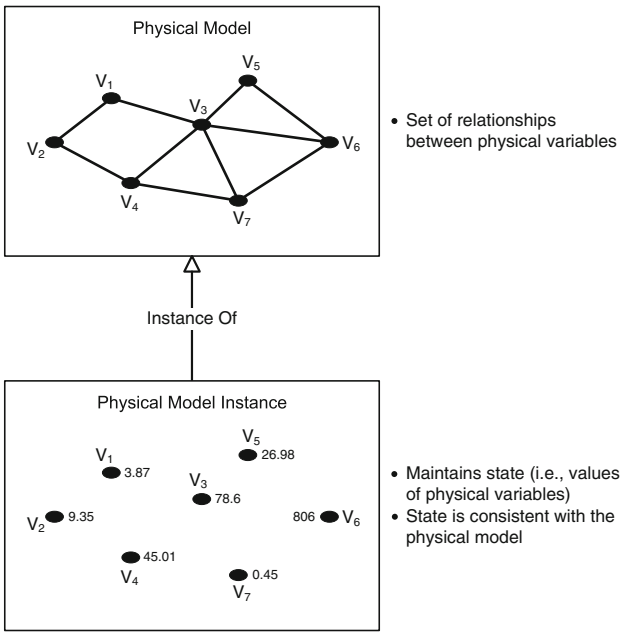

Fig. 8 Instantiation of a physical model

ical model instance uses solving algorithms similar to the algorithms implemented in 20-Sim and Matlab Simulink. For details about these algorithms, we refer the reader to [47]. In this paper, we will suffice with a discussion of the differences between the solving algorithm used by the physical model instance and the solving algorithms implemented in 20-Sim and Matlab Simulink. This discussion is given in Sect. 9.5.

6 Composition using the Composition Filters model

In the previous section, we explained how physical models are represented in software. In this section, we describe how physical models interface with other software modules using the *Composition Filters model*. The Composition Filters model is implemented in the language *Compose**. A short introduction in the Composition Filters model and Compose* is given in Appendix A. For further information on the Composition Filters model and Compose*, we refer to [40,48,52,53].

6.1 Composition overview

In general, artifacts on which the Composition Filters model is applied consist of two elements:[2]

- An *implementation object*, which contains the internal semantics of the artifact (e.g., the instance variables and implementation of methods when the artifact is an object in an object-oriented language).
- An *interface*, which 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).

In the Composition Filters Model, *composition filters* can be attached to the interface of the artifacts. These composition filters provide selection, filtering and transformation operations on interactions (e.g., messages or events) with the interface. Attaching composition filters to an artifact is called *superimposition*.[3] The artifact consisting of the implementation object and the interface, together with the attached composition filters is called a *concern instance*. Figure 9 applies these concepts on physical model instances.

6.1.1 Implementation object of physical model instances

The implementation object of physical model instances contains the physical relationships defined in the physical model and the physical state that is part of the physical model instance.

6.1.2 Interface of physical model instances

To compose physical model instances with software modules, physical model instances provide an interface that is divided into two parts:

- A *base interface* to request and update values of variables in the state of the physical model instance. The base interface is accessible from software modules by sending messages to the physical model instance. The Composition Filters modeldirectly applies to the base interface; filter modules containing input filters can be superimposed on a physical model instance to filter messages that are send to the physical model instance, as shown in the top part of Fig. 9. Section 6.2 describes the base interface in more detail.
- An *event model* that defines *events* that can occur during the execution of a physical model instance. We generalized the Composition Filters modelin such a way that not only messages can be filtered and matched, but also events conforming to a certain *event model*. We defined such an event model for the execution of physical model instances. Examples of events that can occur in the execution of physical model instances are a request of the value of a physical variable, the update of the value of a physical

[2] See also Appendix A for more information about the Composition Filters model.

[3] See also Appendix A.

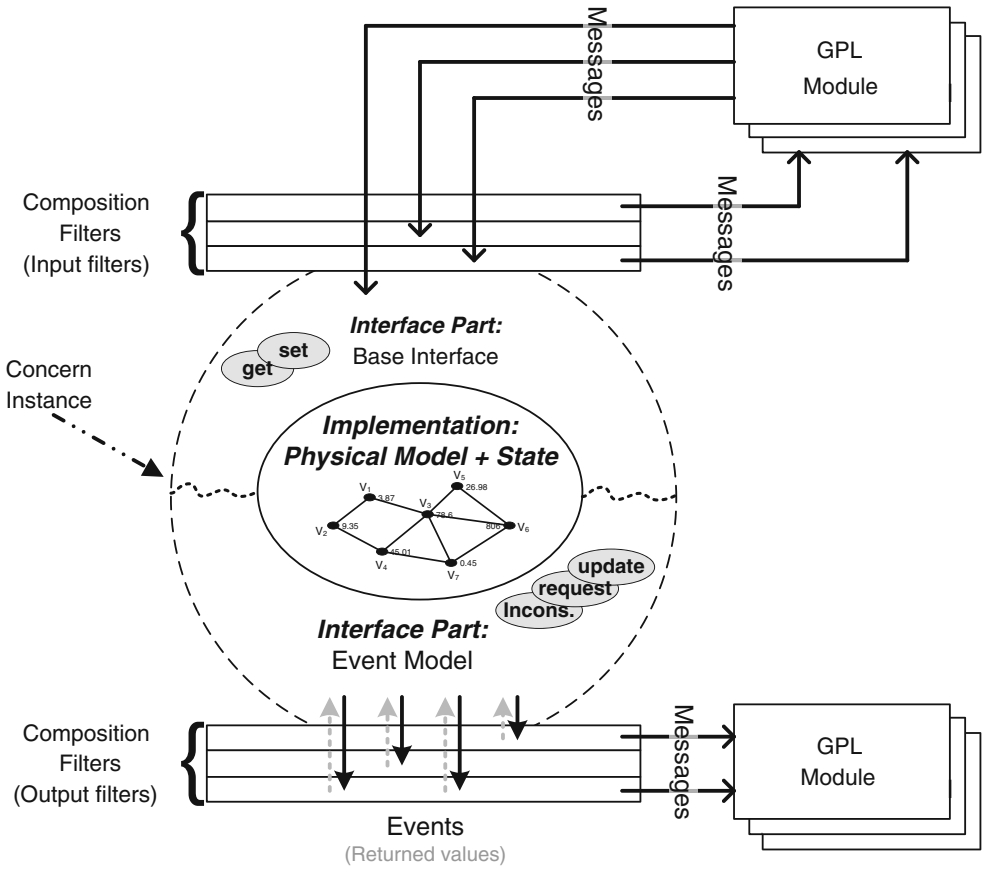

Fig. 9 Composition filters applied to a physical model instance

variable and the detection of an inconsistency in the physical model. Composition filters can be superimposed on physical model instances to filter and match events occurring during the execution of the physical model instance. These composition filters can specify behavior to execute when an event is matched. This behavior can for example be to dispatch a message to a software module to execute certain computational logic. The bottom part of Fig. 9 shows the *event model* interface part and the application of composition filters. Section 6.3 describes the base interface in more detail.

6.2 Base interface

The physical model instance is in an object-oriented language accessible as an object. This object has an interface to obtain values from the physical model instance and to change values in the physical model instance. Furthermore, the interface contains a method to start the evaluation of the physical model instance. This enables the designer to control when a physical model instance is evaluated (e.g., as part of a control loop). Figure 10 shows the base interface of physical model instances.

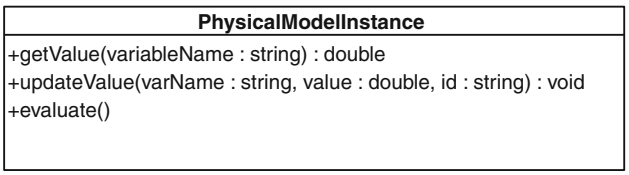

Fig. 10 Base interface of physical model instances

6.3 The event model

The event model defines the types of events that can happen within the execution of a physical model instance and that can be matched within a composition filter. It is a generic definition for physical model instances, not specific for a particular physical model instance. Events have certain properties.

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

6.3.1 Property: eventType

The `eventType` property conveys the type of the event. The possible types are:

Table 1 Event properties

Event properties	
$eventType$	
$variableName$	
$value$	
$returnValue$	
$returnIdentifier$	
$values$	Only defined if $eventType = \text{`}Inconsistency\text{'}$
$margin$	
$enforceReturn$	

- *Request* The value of a variable is requested using the base interface of the physical model instance.
- *Update* The value of a variable is updated using the base interface of the physical model instance.
- *CheckUpdate* At the start of evaluation of a physical model instance, for each variable it is checked whether the value has been updated. This eventType indicates this check and provides the possibility to define behavior that updates the value.
- *Change* The value of a variable has changed during the evaluation of the physical model instance. This type of event occurs after the evaluation algorithm, when the current state is changed to reflect the new state.
- *Inconsistency* An inconsistency has been detected during the evaluation of the physical model instance. Inconsistencies occur when there are multiple ways to determine the value of a physical variable. If the multiple outcomes are not equal, this leads to an *Inconsistency* event.

6.3.2 Property: variableName

This property contains the name of the variable that is the subject of the event.

6.3.3 Property: value

This property contains a value, which differs based on the type of the event:

- *Request* and *CheckUpdate* The current value of the variable in the physical model instance.
- *Update* The updated value of the variable, as provided using the call to the base interface.
- *Change* The new value of the variable. Actually, this is the current value of the variable in the physical model instance; the current value changed during the evaluation of the physical model instance.
- *Inconsistency* One of the multiple (inconsistent) values for the variable, which are determined by the evaluation algorithm. The single value is non-deterministically selected.

6.3.4 Property: returnValue

Can be used by filter actions to return a value. The semantics of this returned value differs based on the type of the event:

- *Request* A `getValue` call has been made to the base interface of the physical model instance, triggering this type of event. Normally, the returned value of a `getValue` call is the current value of the variable. But using the `returnValue` property, a composition filter can change the returned value. The current value of the variable in the physical state is not affected.
- *CheckUpdate* The property can be used to return an updated value of the corresponding physical variable. This updated value will be used by the evaluation algorithm to update the physical state.
- *Update* The property can be used to change the updated value provided by the update-call to the base interface that initiated the event.
- *Change* The property is not used.
- *Inconsistency* The property contains the value that should be used in the physical state (to resolve the inconsistency).

6.3.5 Property: returnIdentifier

This property is only defined for events with type *CheckUpdate*. It can be used to provide an identifier for the update.

6.3.6 Property: values

In case the event type is *Inconsistency*, this property contains all values for the corresponding variable that are derived by the evaluation algorithm. The property provides a mapping from a `String` identifier to the corresponding value. The identifier indicates the source of the value.

6.3.7 Property: margin

In case the event type is *Inconsistency*, this property contains the largest difference between the values in the property `values`. The `margin` property can be used, for example, to filter *Inconsistency* events for which the difference between the values is larger than a certain threshold value.

6.3.8 Property: enforceReturn

In case the event type is *Inconsistency*, this property contains a Boolean value indicating whether the value in the property `returnValue` should be enforced upon the state

or not. Enforcing a value means that the values of other variables are made consistent, according to the physical model, with the provided value in the `returnValue` property.

6.4 Example: defining composition filters

Listing 7 shows an example composition filters specification. This specification is applied to the physical model instance of the Drum Shuttling case studythat contains the drum rotation model specified in Sect. 5.2.1.

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

```
1  filtermodule RotationIO{
2    externals
3      pcSensor: Sensor =
           IO.getPulseCountSensor();
4      shControl: ShuttlingController =
           Control.getShuttlingController();
5    outputfilters
6      pcFilter: Dispatch =
           (event.variableName=='pulseCount'
           & event.eventType=='CheckUpdate')
7        {msg.target=pcSensor;
             msg.selector='getValue'};
8      xPosFilter: Dispatch =
           (event.variable=='xPosition' &
           event.eventType=='Change')
9        {msg.target=shControl;
             msg.selector='setXPos'};
10 }
11
12 superimposition{
13   selectors
14     models = { M | isModelWithNameInList
           (M, ['motor', 'gears', 'drum']) };
15   filtermodules
16     models <- RotationIO;
17 }
```

Listing 7 Composition filters specification to handle drum rotation

Lines 1 to 10 show the definition of the filter module `RotationIO`. 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 $CheckUpdate$ events concerning variable $pulseCount$. After the matching part, this filter has an assignment part (shown on Line 7). In this assignment part, the target and selector of the to be dispatched 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 `resultValue` property. This value is used by the evaluation algorithm to update the physical state.

Lines 6 and 7 show the definition of the second composition filter, which matches on $Change$ 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`.

6.5 Default composition behavior

When an event occurs in the execution of the physical model instance, the value of the physical variable that is the subject of the event may be influenced. For example, if the event type is $Update$, the update value provided to the base interface with the `updateValue` call can be changed by a composition filter. If the event type is $Inconsistency$, the proper value should be selected. Using the `returnValue` property, a composition filter can give a new value for the physical variable. But if no composition filter has been defined to provide a return value, there is a default behavior that is performed. The default behavior is to return the current value in the physical state as the new value (`event.returnValue = event.value`).

7 Analysis and verification

In this section, we will describe domain-specific static analysis and runtime verification that is possible with our approach.

7.1 Domain-specific analysis

Combining a DSML with a GPL provides the possibility to apply the domain-specific analysis techniques that come with the DSML. Examples of analysis that can be performed on SIDOPS+ specifications are checking of unit consistency of physical relationships (if units are defined for physical variables) and detection of algebraic loops.

7.2 Runtime verification of physical models

The physical models that are combined with GPL modules might not correspond with physical reality, for reasons that include:

- The underlying assumptions about physical reality on which the physical models are based might not be accurate enough.
- The system might be used in different operational conditions than considered during design.
- Physical characteristics might change over time, because of, for instance, wear and tear of physical components in the system.
- Physical characteristics might change because of evolution of the physical system.
- The engineer implementing a physical model might introduce a fault.

As inconsistencies between the physical model and physical reality can lead to failures in the behavior of the system, such inconsistencies need to be detected. Because the system cannot be tested in all possible circumstances, runtime verification of the physical models is necessary.

7.2.1 Detecting inconsistencies at runtime

In [50], we described a technique to perform runtime verification of physical models. This technique makes use of redundancy in the physical model (i.e., physical variables that have multiple sources for their values, such as multiple sensors and relationships). Inconsistencies in the different values for these physical variables can 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 8 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. Lines 6 to 11 show that the filter module is superimposed on all physical model instances in the system.

```
1  filtermodule inconsistencyLogging{
2    outputfilters
3      inconLog: Logging =
           (event.eventType=='Inconsistency'
           & event.margin > 0.1);
4  }
5
6  superimposition{
7    selectors
8      models = { M | isModelInstance (M) };
9    filtermodules
10     models <- inconsistencyLogging;
11 }
```

Listing 8 Composition filters specification to log inconsistencies

7.2.2 Monitor wear and tear

One advantage of the runtime verification approach is that it becomes possible to monitor wear and tear in the system. We will now show an example of this.

Over time, the radiator in the Warm Process case study may get polluted and therefore less efficient. If not detected early enough, this can cause damage to the system. Using runtime verification of physical models, early detection has been implemented. Listing 9 shows (partial) SIDOPS+ specification for the warm process case, described in Sect. 2.1. We added a second equation, to relate T_{contact} back to T_{belt}. In this way, T_{belt} has two sources of values: the sensor and the newly added relationship.

```
1  [Model radiator.phys]
2  ...
3  equations
4    Tcontact = c4 * Prad / sqrt(v) + Tbelt
5
6  [Model belt.phys]
7  ...
8  equations
9    Tbelt = c5 * (Tcontact + c6 * Tph) *
         sqrt(v)
```

Listing 9 Warm process SIDOPS+ specification

In a new system, the two sources provide the same values. But when the radiator gets polluted, the two values begin to differ. This can be detected by the composition filter specification shown in Listing 10. Line 3 shows a `ServiceCall` filter, that performs a service call when it accepts (in this case when the margin becomes larger than 0.2).

```
1  filtermodule radiatorWear{
2    outputfilters
3      radWear: ServiceCall =
           (event.variableName='Tbelt' &
           event.eventType=='Inconsistency' &
            event.margin > 0.2);
4  }
```

Listing 10 Composition filters specification to detect radiator wear

7.2.3 Monitor acceptable ranges

Listing 11 shows an example that does not check for inconsistencies, but monitors whether a certain physical variable stays within an acceptable range. 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 has changed. 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.

```
filtermodule TphMonitoring{
  outputfilters
    tphMonitor: Logging =
        (event.variableName=='Tph' &
        event.eventType=='Change' &
        (event.value < 60 | event.value >
        100));
}
```

Listing 11 Composition filters specification to log inconsistencies

7.3 Composition analysis

If a physical model is composed with GPL modules, there should be enough input/output interaction between the model and the modules, so that the model can correctly compute the desired output. For example, if the model from Fig. 7 is composed with GPL modules to supply the $xPos$, but there is no interaction between the model and a module that can provide $pulseCount$ (e.g., through a sensor), then $xPos$ can never be correctly computed.

We can detect this problem in the following way. First, the composition between the model and the GPL modules is checked to detect for which variables input is provided to the model, and which variables are requested as output from the model. Then, we can check whether each variable that is requested as output is solvable (meaning that there is a way to compute its value). The algorithm to perform this operation is explained in detail in [47].

8 Evaluation of the approach

Our approach enables the composition of physical models specified in a DSML with other software modules implemented in a GPL. This provides the benefits of using a DSML to specify physical models with the freedom of a GPL to implement other functionality. In this section, we illustrate with a number of evolution scenarios the benefits of our approach concerning the maintainability and evolvability of the physical models implemented in embedded control software.

8.1 Evolution scenarios

Using relevant evolution scenarios, we explain how our approach reduces programming effort. Several evolution scenarios have an impact on the implemented physical relationships. But there are also evolution scenarios that have an impact on the GPL modules. The impact of evolution scenarios on the physical relationships can be categorized as follows:

1. *Introduction* A new physical relationship is added.
2. *Elimination* A physical relationship is removed.
3. *Change* An existing physical relationship is changed. There are several possibilities:
   (a) *Constant value* The value of a constant in the relationship is changed.[4]
   (b) *Structural* A change to the structure of the relationship.
   - *Refinement* A constant in the relationship is replaced by a subformula introducing new variables in the relationship.
   - *De-refinement* Replacing a subformula by a constant.
   - *Splitting* A relationship is broken into smaller parts. i.e., subformulas are extracted to make the (implicit) variable value that they specify explicit. In the original relationship, this subformula is replaced by the variable name that it expresses.
   - *Merging* The reverse process of splitting.

8.1.1 *Scenario A: addition/removal of a component to/from the physical system*

Description The system consists of multiple components. It is likely that during the lifecycle of a product type a specific component is added or removed from the design, for example to extend its functionality or to lower costs.
Example The removal of the paper heater in the warm process case study. Heating will then be entirely done by the radiator.
Impact Components in the system often have a direct correspondence with modules that implement control logic. For example, the paper heater has corresponding control logic, which is implement in the `Paper Heater Controller` module. So, adding or removing a component from the system can lead to the addition or removal of a module. Besides that, it can also lead to introduction/elimination of a physical relationship, because a value of a variable, necessary to control the component, should be derived. Or it can lead to refinement/derefinement of relationships because a variable that was constant now becomes variable due to the addition of a component.
In the traditional approach, because the affected relationships are spread through the code of different GPL modules, it might be hard to locate them, giving higher risk of introducing errors.
With our approach, the relationships are made explicit and are separated from the GPL modules. Therefore, affected relationships are now easier to find and updates

[4] The word 'constant' refers in this case to a fixed value in the physical relationship for one specific design. Therefore, if the design changes, the value of constants can also change.

are localized to the SIDOPS+ specification. This reduces programming effort.

8.1.2 Scenario B: change of the physical structure of the system

Description With physical structure is meant the location and arrangement of the different components of the system. This is subject to change during the lifecycle of the product type.
Example The position of the paper heater in the paper path.
Impact The implemented physical relationships are based on the existing physical structure of the system. Therefore, a change of this structure has an impact on the implemented relationships, especially on the constants within these relationships. For example, the change of position of the paper heater has an impact on the paper temperature drop between the paper heater and the contact point. This affects Eq. 1.
In the traditional approach, this relationship is implemented in a different module (the `Radiator Controller` module), not directly related to the changed item in the system. This makes it harder to locate the affected code, giving a higher risk of introducing errors.
In our approach, the scenario has a direct impact on the explicitly defined relationship in the SIDOPS+ specification. Because the relationship is explicitly defined, it is easier to locate, thus reducing programming effort.

8.1.3 Scenario C: change in operating conditions of the system

Description During the design of the system, certain operating conditions, like temperature of the environment, are assumed to be (approximately) fixed at a certain level. However, during the lifecycle, the assumed operating conditions might change.
Example The printing system is first targeted at a market in a moderate climate. Later on, the product becomes targeted at markets in a hot climate, which means a higher environmental temperature.
Impact The fixed conditions are abstracted away in the constants of certain relationships. A higher environmental temperature has an influence on the temperature loss of the paper, affecting constant $c1$ in Eq. 1. It also has an impact on the pinch temperature, affecting constant $c1$ in Eq. 2.
From this example it is clear that the change of operating conditions can have varying impact on different physical relationships, which are in the traditional approach spread over the control software. Localizing the affected parts is therefore harder, which can lead to the introduction of errors.
In our approach, the impact is limited to the SIDOPS+ specifications instead of affecting GPL modules. Programming effort is reduced.

8.1.4 Scenario D: more variety in operating conditions

Description Certain operating conditions could change from being fixed to being variable.
Example An example of this can be the introduction of different paper weights. At first, the system was designed to handle only one paper weight. But later on, to target new customers, the system is adjusted to handle different paper weights.
Impact The impact of this change is that the variable m_{paper} is now going to vary, so becomes important in Eq. 1. This means that this variable, which was first abstracted away in constant $c2$, is introduced into the equation. The new equation will be something like:

$$T^{\text{SP}}_{\text{contact}} = c1 \cdot v - \frac{c2' \cdot T_{\text{ph}}}{\sqrt{m_{\text{paper}}}} + c3$$

In general, the connection between the varying conditions and the affected relationships might not be apparent, and the relationships might be scattered through the GPL modules. This makes it harder to locate these relationships, increasing the potential for errors.
Now that the relationships are explicitly defined with the SIDOPS+ approach, this impacted relationship is easier to find and the change does not affect the GPL modules that implement control logic anymore. Thus, programming effort is reduced.

8.1.5 Scenario E: the increase or decrease of available information

Description The available information to the control software might increase or decrease, because sensors are being added to or removed from the system, or new variables and their relationships with other variables become known.
Example A sensor that measures temperature of the environment is added. This variable influences Eq. 1, but has been neglected before as stochastic variation.
Impact Now that we have the information available, it can be used in Eq. 1. This means the introduction of a new variable in this relationship. Equation 1 changes in the following way:

$$T^{\text{SP}}_{\text{contact}} = c1 \cdot v - c2 \cdot T_{\text{ph}} + c3 + c5 \cdot T_{\text{environment}}$$

Impacted relationships might be scattered through the GPL modules.
As in the previous scenario, with the SIDOPS+ approach this relationship is easier to locate and changes do not impact the GPL modules anymore.

8.1.6 Scenario F: changing a control component

Description The control behavior of a control component might be changed.
Example In the drum-shuttling case the control behavior of the `ShuttlingController` changes.
Impact The `ShuttlingController` logic is not tangled with physical relationships anymore. Therefore, changing it is easier. Furthermore, the component can more easily be reused in other printing systems, as it is not tightly coupled with the stepper motor and the drum motor anymore, but has a clear interface to x-position and z-position. Higher cohesion of the module and more potential for reuse have a positive impact on programming effort.

8.1.7 Scenario G: dynamically adapting a physical relationship

Description A physical relationship might change during runtime, due to adaptive behavior of the system.
Example Equation 1 is different if the printing system is starting up or is idle.
Impact Although not investigated in this paper, now that the physical relationships are made explicit, it becomes easier to build in language mechanisms to change them at runtime. If the relationships are implemented in the GPL, GPL constructs, such as `if-else` statements, are necessary to provide this adaptive behavior.

From the analysis of these relevant evolution scenarios we conclude that changes in the physical system can have an impact on physical relationships, which are often spread through the software implementation. Because of the implicit implementation, the developer might not even be aware of the relationships and of the changes that should be made to them. This can be a source of errors. Also locating the physical relationships and changing them can be costly.

Using our approach to specify physical models in the SIDOPS+ language and compose them with GPL modules, the physical relationships are explicitly specified. This makes it easier to locate and modify them, reducing programming effort.

9 Discussion

In this section, we discuss some additional aspects of our approach.

9.1 Applicability on real-time systems

An important aspect in the design of embedded software is dealing with real-time functionality [38]. Therefore, programming languages and tools aiming at embedded software should support the creation of software that is able to meet timing constraints under all circumstances. This support includes efficient software execution, software execution that is deterministic in time and memory and the ability to precisely analyze the software with respect to its timing behavior.

The implementation of our approach contains a runtime infrastructure with an interpreter to execute the SIDOPS+ models and the composition filters. Such an interpreter-based approach raises the question whether the resulting execution of the embedded software is efficient and deterministic in time, i.e., whether the embedded software is able to meet its timing constraints under all circumstances. In this paper, we did not address this concern. We mainly focused on combining a DSML for physical modeling with a GPL, to improve the quality characteristics *Maintainability* and *Reliability*. With respect to these quality characteristics, an interpreter-based execution environment is sufficient to demonstrate our approach.

To enable industrial application of our approach, efficiency and deterministic operation are definitely concerns that should be addressed. Efficient compilation instead of applying an interpreter is part of our future work. Efficient compilation algorithms for aspect-oriented languages, such as the Composition Filters model, are known in literature, e.g., in [14,46,48]. The 20-Sim tooling contains code generators to compile 20-Sim/SIDOPS+ models to efficient and deterministic C code [1]. Future work involves how these different compilation approaches can be combined, to be able to efficiently compile the composition (using the Composition Filters model) of the SIDOPS+ models with the GPL modules.

9.2 Separating design rationale from control logic

The design of a controller depends heavily on the physical characteristics of the system being controlled. Control engineers study these physical characteristics and decide on the type of controller to use and the parameters for this controller, to obtain certain desired characteristics, such as stability, small error margins, reaction time, etc. As such, these

physical characteristics are part of the design rationale of the control logic.

A variety of work has been performed on specifying control logic on a higher abstraction level, independent of specific physical characteristics of the system. The actual control logic is then generated from the higher-level specification and a model of the physical characteristics. In this way, the specification of the control logic is less vulnerable for changes in the physical system. An example of such work is the work on supervisory control synthesis by Rooda et al. [41,51].

The described method in this paper does not aim at specifying and separating this design rationale from the control logic. Also, this paper did not introduce new types of control logic. This paper presented techniques to modularize and compose models of physical characteristics that are being applied in the control logic, as computational functionality, not as design rationale of the control logic. Control engineers still decide on which models of physical characteristics are part of the computational logic and in what ways these models interact with other control modules. The models of physical characteristics are for example used to model the actual system state, to model a desired system state or to model constraints on the state of the system.

9.3 Control logic in a DSML

In this paper, we made a distinction between physical characteristics and control logic. Physical characteristics describe aspects of the physical system that are valid independent of the control software, such as a natural relationship between certain physical variables (e.g., the relationship between T_{ph}, T_{contact} and v that defines acceptable print quality). Control logic are those computations designed to actively influence the state of the system, such as a feedback controller that calculates a certain output signal based on some input signals and the controllers state.

Besides being applied to model physical systems, toolsets such as 20-Sim and Matlab Simulink can also be applied to model continuous control logic. The computational model for continuous control logic is similar to the computational model for physical characteristics. For example, in the SIDOPS+ language, continuous control logic can be specified using equations. Therefore, our approach can also be applied to SIDOPS+ models containing continuous control logic. This provides the additional benefits of the Composition Filters model, such as the possibility to implement aspect-oriented functionality on continuous control logic.

In this paper, we did not investigate and utilize the capability of the 20-Sim toolset to model the control logic. We only used the 20-Sim toolset for the domain-specific modeling of the physical characteristics being applied in the control software. This was done to illustrate the possibility to model certain concerns of the control software with a DSML and to illustrate that the models specified in a DSML can be composed with modules specified in a GPL that implements other aspects of the system. If we also applied the DSML to model the control logic, the amount of GPL code in the example cases would be limited. However, this is not the case in general; the limited amount of GPL code is caused by the fact that the example cases are limited, and only contain a subset of the concerns of realistic control software. Examples of concerns that are part of realistic control software are management of system states (e.g., idle, start-up, available), scheduling of (discrete) tasks, recovering from errors, monitoring the available resources (e.g., the amount of toner, number of sheets of paper in the paper tray), processing of user input, security, communication, maintenance tasks, etc. These concerns were left out of the example case studies, as they would require and extensive introduction of the example case studies and of the domain knowledge needed to understand them. Toolsets such as 20-Sim and Matlab Simulink are not designed to handle these concerns. Therefore, still a substantial amount of GPL code is needed.

9.4 Risks of physical model decomposition

In this paper, we presented techniques to modularize physical models used in control software and to compose the physical models with other software modules. The techniques ensure that a given modularization and composition is correct from a language perspective (i.e., it results in executable software). However, this paper did not attempt to provide techniques that ensure the correctness of the modularization and composition of physical models from the perspective of the application. It is up to the engineers of the system to ensure the correctness of the modularization and composition from the application perspective.

If the modularization and composition of physical models in software is not performed with care, undesirable side effects can occur in the behavior of the control software and as such in the behavior of the physical machine. Examples of undesirable side effects are unforeseen interactions between physical models or between a physical model and a GPL module and deviation from desirable behavior. These problems are related to modularization, as the internal working of a module can be hidden for the developer of another module. If the interfaces of the modules are not properly specified, unforeseen interactions can occur. Note, however, that these issues concerning modularization are general and not specific for the modularization of physical models in control software.

This paper discussed a number of verification techniques that can be used to detect problems in control behavior (e.g., the value of a certain physical variable goes outside its

limits). The cause of such a problem may be diagnosed as a modularization issue.

9.5 Difference between 20-Sim simulation and physical model execution

The equation solving algorithm used by physical model instances is largely the same as the procedure used in 20-Sim. However, there are some differences. These differences are caused by the fact that the execution of physical models in software relies on externally provided updates of physical variables. We introduced the possibility to have multiple ways to determine a value, possibly leading to inconsistencies in the physical model. We added a mechanism to solve inconsistencies. However, the possibility of having multiple ways to determine a value leads to differences in which algebraic loops are solved and to differences in the execution order of equations.

9.5.1 Algebraic loops

Because there are multiple ways to determine a value, the handling of algebraic loops is different:

- If the value of one of the variables (referred to as variable a) in the loop is also provided from another source (e.g., a sensor or a different physical relationship not part of the algebraic loop), then the loop is solved using this value. This leads to multiple values for the variable a, because of the loop. This way of solving can be emulated in 20-Sim in the following way:
  - Replace the physical variable that can have multiple values with multiple physical variables, one for each way to determine the value (thus breaking the loop).
  - Add logic to compare the values of the multiple physical variables derived from the original physical variable.
- Otherwise, the loop is solved using methods also applied by 20-Sim (not implemented in our current tooling).

9.5.2 Evaluation order

Because there can be multiple sources for values, the evaluation order of equations can be ambiguous. To aid the designer, we implemented a specific evaluation order, giving preference to forward solving of equations before backward solving.

Replacing a physical variable that can have multiple values with multiple physical variables eliminates this ambiguity. In this way, the 20-Sim execution method can be used.

9.6 Dynamic adaptation of physical models

A physical model instance is created from one or more physical models. In case multiple physical models are used, each model forms a submodel of the implicitly composed physical model that expresses the physical relationships of the instance.

The application of physical model instances can be made more flexible, if the physical model it refers to is flexible. If it is for example possible to add, remove, replace submodels from the physical model, the behavior of the physical model instance can vary at runtime, to reflect for example discrete changes in the physics of the system or changing constraints. An example of changing constraints is the constraint between T_{contact} and T_{ph} in the warm process case. The constraint presented by Eq. 1 is only required in the running state of the printing machine. If the printing machine is for example in the idle or startup state, the constraint is different. By making it possible to manipulate the physical model at runtime, such examples of adaptive behavior can be easily implemented.

9.7 Recovery actions after runtime verification

If a failure has been detected using runtime verification (meaning that the different redundant values of a certain physical variable are not consistent), 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.
- Use the value with the highest confidence level. For example, a voting scheme can be used if there are more than two ways to determine the value of the parameter.
- Log the inconsistency and chose one of the values, either randomly or by some preference.

10 Related work

10.1 Domain-specific models in embedded control software

Domain-specific models are commonly used in the development of embedded software. For example, the Giotto approach [31] is a domain-specific language that aims to separate timing from functionality in embedded control software. A timing specification, called Giotto timing program, can be generated from a simulink control model. The timing program supervises the functionality programs, which implement the control functionality. The timing program is independent of a given implementation platform, supporting portability between different platforms. The Giotto approach

provides tooling to verify whether a given implementation platform can handle the timing constraints given in the Giotto timing program. The Giotto approach is comparable to our approach as it also applies a DSML to implement part of the computational logic for embedded control software. Interesting in the Giotto approach is that the DSML used is a DSML to model physical characteristics and continuous control logic, i.e., a DSML similar to the SIDOPS+ language. In this way, the Giotto approach can be an addition to our approach.

There are approaches that apply aspect-oriented techniques to domain-specific modeling approaches. An example of this is the C-SAW approach [29]. C-SAW provides a technique and tooling to add aspects to software models, which are higher abstraction levels of the software implementation. This differs from our approach in two ways. The first difference is that the models referred to in our approach are not software models, but models of physical characteristics (in general, models of domain concepts). These models of physical characteristics are applied in control software to provide part of the computational logic, which is realized by adding execution semantics to the modeling language. The second difference is that our approach does not aim to express aspects in the domain-specific models themselves, but uses aspect-oriented composition technology to compose the models in the DSML with the modules in the GPL at the abstraction level of both languages.

10.2 Interaction-based approaches

One usage of physical models in embedded control software is to implement the interaction between different control modules. In this section, we compare our approach to existing interaction and coordination approaches.

Contracts [30,32] were introduced to explicitly model interactions among a group of objects. These constructs capture behavioral dependencies specified with a set of preconditions and invariants [30]. The contract principle was also applied to component-based systems, to extend component interfaces with behavioral constraints [10]. Hereby, contracts are used to check the compliance of components. Similarly, contracts that provide formal semantics regarding object interactions enable conformance checking at compile-time [30]. As such, these approaches support maintenance and reuse of software systems. In our work, we use the formal specification not only for domain-specific analysis and checking for compliance but we also compose the specified models with other software modules and execute them. So, the specification becomes a part of the implementation.

Meta-object protocols (MOP) [33] have been introduced as supplemental constructs to programming languages by means of which the language's behavior can be modified. As such, semantics of a program becomes open and extensible. MOPs, as well as reflection [39] mechanisms, could be used to explicitly model the interaction between modules that is caused by the physical characteristics. In our work, we have proposed a domain-specific solution to specify these physical characteristics in a declarative way and to support domain-specific analysis. Furthermore, we apply the aspect-oriented Composition Filters modelto compose DSML models with software modules. Such aspect-oriented techniques can be supported using reflection and meta-object protocols [15].

There have been efforts to abstract away and encapsulate the coordination among a number of computational entities (e.g., objects, components, etc.). These efforts led to coordinated behavior abstractions [6], and later on to coordination models and languages [43]. These approaches improve reusability by enabling explicit modeling of interaction, enforcement of invariant behavior, and separation of interaction details from the computational concerns. Modeling coordination is not the aim of this work and as such we have not proposed an alternative model or language for this. Instead, we have introduced a modular extension to state-of-the-practice languages for explicitly modeling interaction caused by physical characteristics. The models that deal with the specified interaction are composed with the existing GPL modules using the Composition Filters model.

Service orchestration is used to compose multiple (web-) services in a meaningful way to create a larger application. The orchestration language deals with issues as sequencing of the services to call, parallel calls to different services, branching in the services to call, etc. Examples of languages to perform service orchestration are BPEL [2] and Orc [35]. Service orchestration is specifically aimed at coordination among services. Our approach can capture interaction caused by physical characteristics, using domain-specific abstractions.

10.3 Connection with system modeling approaches

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. There are several approaches to integrate such models and describe the system from different perspectives so that consistency can be maintained and changes in one model can be automatically reflected in other models. Example of such approaches are the Knowledge Intensive Engineering Framework (KIEF) [56] and the Speeds approach [44]. Such efforts can be extended to include software specifications, because software functionality is heavily based on aspects of the physical system, such as the structure and the physical interaction between components. Furthermore, such an integration can provide traceability between the physical relationships and the corresponding structures in

the system models. This can be helpful in case of evolution, as the impact of changes in the system to the physical relationships can be automatically traced within the embedded control software.

10.3.1 KIEF

Forbus describes in [25] the concept of qualitative process theory. This is the analysis and specification of the qualitative relationships between different physical variables 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 variables [56]. They can be derived from other system models, like structural models.

Although the parameter networks do not describe quantitative relationships, they can be used as a starting point to derive the quantitative relationships needed to create the physical models. First, the parameter network can be used to check whether certain variables are solvable. Secondly, if the quantitative mapping is possible, the part of the parameter network that reflects this mapping should then be quantified to create the physical model.

10.3.2 Speeds

The Speeds approach [44] is an embedded system design methodology that supports the composition of heterogeneous subsystems using semantic-based modeling methods. To compose different models of the system, the approach defines the concept of *heterogeneous rich-component* model that can represent different functional and architectural abstractions in embedded system models, such as timing and safety properties. Although the Speeds approach offers composition of different models, the Speeds approach differs from our approach as it provides an integrated approach for embedded system modeling instead of an approach to apply such heterogeneous (domain-specific) models in embedded control software and to compose these models with GPL software modules.

The integration of the Speeds approach with our approach is interesting future work; this integration can provide composition between heterogeneous domain-specific models in embedded control software, in addition to the composition of DSML models with GPL modules.

10.4 Heterogeneous composition of computational models

The Ptolemy approach [22] provides a way to model *embedded computational systems* (not necessarily limited to software) that consist of heterogeneous types of components (i.e., types of components that differ in how they communicate and interact). Arbitrarily composing heterogeneous components can lead to ambiguities in the interaction between the components (due to the differences in the way these components communicate and interact), resulting in unexpected emerging behavior. To prevent such problems, the Ptolemy approach uses hierarchical nesting of different types of components, resulting in a homogeneous composition at each hierarchical level.

The basic building block in the Ptolemy approach is called an actor. Actors can communicate with each other through ports. A system of multiple communicating actors can be encapsulated into a higher-level actor. In this way, hierarchical structures can be created. The way data flow and control flow between actors is performed at a certain hierarchical level is not defined by the actors themselves, but by a separate model of computation (MoC). Different composite actors can have different MoCs. Some MoC implementations (also called domains) that have been realised are: Communication Sequential Processes (CSP), Continuous Time (CT), Discrete Event (DE), Process Network (PN) and Synchronous Dataflow (SDF).

Our approach of combining the semantics of the DSML for physical models with the semantics of the GPL is similar to the Ptolemy approach. The semantics of physical models is encapsulated in the physical model instance. This semantics is well-defined and there is no interference with the semantics of the GPL. The modules in the GPL are oblivious to this internal semantics of the physical model instance. The interaction between the physical model instance and the modules specified in the GPL is performed using the base interface and the Composition Filters model, which is natural from the perspective of the GPL. This mechanism can be seen as two layers of hierarchical heterogeneous composition from the Ptolemy approach (the physical model semantics is encapsulated into physical model instance actors, which can be composed with other GPL modules/actors). This prevents ambiguity in the control flow and data flow, thus preventing unexpected emerging behavior.

10.5 Runtime verification

Literature shows, e.g., in the taxonomy of Delgado et al. [19] and in the work of Barringer et al. [9], 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 [18] and the tracematches extension to AspectJ [7]. Such approaches could not be applied in our

case, as we needed 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, we needed a new approach in which we use redundant models of physical relationships to verify the conformance with physical reality.

van Gemund et al. [57] have worked on fault diagnosis in embedded systems. 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 [45] and de Kleer [36], 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 [57]. van Gemund et al. [3,23] combined both approaches to be applied on the combination of embedded system and the corresponding embedded software.

The work of Bapty et al. [8] facilitates dynamic reconfiguration of systems for error recovery. Hereby, all the development activities are performed at the modeling level, with various models focusing on different aspects of the system. This is achieved by adopting the so-called model-integrated computing (MIC) approach [8], which requires creating a development environment that is customized for a specific application domain.

11 Conclusion and future work

This paper presented, as a structuring and implementation method in embedded software design, an approach to compose models of physical characteristics specified in a domain-specific modeling language (DSML) with software modules specified in a general-purpose programming language (GPL) at the abstraction level of both languages. As such, this approach combines the benefits (e.g., ease of realization, maintainability, reusability) of a DSML to model physical characteristics with the freedom of a GPL to implement the application-specific functionality of the control software. The SIDOPS+ language from the 20-Sim toolset is used as the DSML to model physical characteristics. The composition filters approach is applied to compose DSML models with GPL modules. The presented approach is implemented using an interpreter-based style and by making a connection with the Compose* toolset.

Using a DSML to model physical characteristics in embedded control software provides the additional benefits of domain-specific analysis techniques, to detect errors and defects that otherwise would have remained unnoticed. A number of domain-specific errors and defects can be statically detected at the DSML level. In cases where static analysis is not possible due to runtime behavior, we offer complementary domain-specific runtime verification techniques. These techniques can detect errors that arise over time, e.g., due to wear and tear of the physical system.

We illustrated the applicability of our approach using two industrial case studies taken from the printing system domain. Using a number of evolution scenarios, we showed the benefits of our approach concerning maintainability.

Based on these observations we conclude that components that deal with the physical characteristics in an embedded control system can be effectively modularized using DSMLs like SIDOPS+. Also, the composition operators provided by the Composition Filters modeloffer a flexible way to integrate DSML models and GPL modules. DSML models can be effectively verified using domain-specific static and dynamic analysis techniques. Our approach offers increased reuse and flexibility, and enhanced readability and reliability in the design of embedded systems.

We would like to extend our work in the following way. We have previously introduced an architectural style [49] for embedded control systems. This style conforms to the component-and-connector view, where some of the components provide the essential control variables through a set of interfaces. However, physical relationships among these variables are currently implicit. We would like to extend this style by explicit modeling of physical relationships at the architecture design level, using our approach presented in this paper.

We would like to perform controlled experiments with our industrial partner, to measure the actual effort reduction and fault prevention that can be achieved with our approach. Furthermore, additional research will be performed on efficient compilation techniques for the approach presented in this paper. Also, the combination with other DSMLs, such as the Giotto approach [31], will be investigated.

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 for reviewing this paper and providing useful feedback.

Appendix A: Background: the composition filters model

In this appendix we give a short introduction in the Composition Filters modeland the Compose* language that implements the Composition Filters model. For further information on the Composition Filters modeland the Compose* language, we refer to [40,48,52,53].

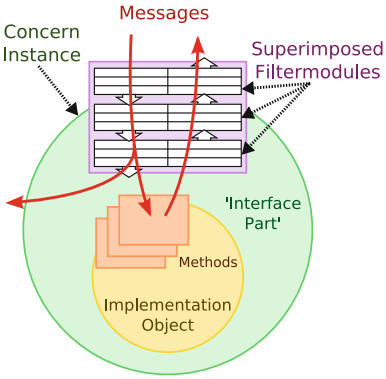

Fig. 11 Overview of the Composition Filters model

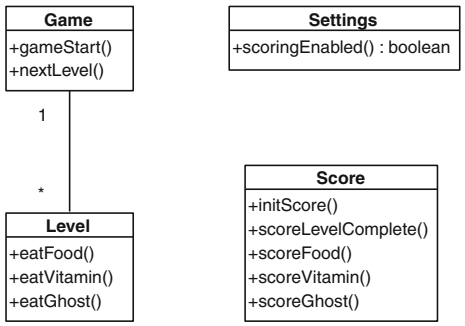

Fig. 12 Some classes in a Pacman game

A.1 Introduction into the composition filters model

A key design goal of the *Composition Filters model* is to improve the composability of programs written in object-based programming languages. The Composition Filters modelhas evolved from the first (published) version of the Sina language in the late 1980s [4,5], to a version that supports language independent composition of crosscutting concerns [12,53].

The Composition Filters modelcan be applied to object-based systems. In such a system, objects can send *messages* between each other, e.g., in the form of method calls or events. In the Composition Filters model, these messages can be filtered using a set of *filters*, as shown in Fig. 11.

Each *filter* has a *filter type*, which defines the behavior that should be executed if the filter accepts the message and the behavior that should be executed if the filter rejects the message. The matching behavior of a filter is specified by *filter expressions*, which offer a simple declarative language for state and message matching. Filters defining related functionality are grouped in so-called *filter modules*. Such filter modules can also encapsulate some internal state or share state with other objects.

To indicate which filter modules should be applied (*superimposed*) to which objects, we use *superimposition selectors*. A superimposition selector selects a set of classes using a Prolog-based selector language. A specified filter module is applied to this selected set of classes. The result is that all messages sent to and received by all instances of those selected classes, have to pass through the filters within the filter module.

The Composition Filters modelcan be applied to many different languages, and we have done so e.g., to SmallTalk [21], Java [55] and C++ [28]. The most recent implementation of the Composition Filters modelis the *Compose** language and toolset, which not only supports .NET, but also Java and C. The next subsection introduces the Compose* language.

A.2 The compose* language

This section introduces the Compose* language using an example in which the Composition Filters modelis applied to implement scoring functionality in a Pacman game.

Figure 12 shows the class diagram of part of a Pacman implementation. The diagram contains the classes `Game` and `Level`, which manage respectively the game and the levels in the game. `Game` contains method `gameStart`, which initializes a new game, and method `nextLevel`, which initializes a new level after a previous level has been completed. The class `Level` contains methods `eatFood`, `eatVitamin` and `eatGhost`, which manage, respectively, Pacman eating a piece of food, Pacman eating a vitamin and Pacman eating a ghost.

The game includes an option to maintain a score. The class `Settings` contains a flag that indicates whether a score should be maintained. Scores are given for various actions of Pacman: eating a piece of food, eating a vitamin, eating a ghost and finishing a level. Furthermore, scoring should be initialized/reset at the start of a new game. The class `Score` contains a method to initialize scoring (`initScore()`) and methods to add a score when a certain action has happened (`scoreLevelComplete()`, `scoreFood()`, `scoreVitamin()` and `scoreGhost()`). Because of the crosscutting nature of scoring functionality with the classes `Game` and `Level`, composition filters are used to compose `Score` with these classes.

Listing 12 shows the composition filters specification that composes the scoring functionality with the Pacman game. The listing shows the definition of the *concern* `ScoringConcern`. This concern consists of one *filter module* definition and one *superimposition* definition.

```
1  concern ScoringConcern in pacman{
2    filtermodule scoring{
3      externals
4        score: pacman.Score =
             pacman.Score.instance();
5        settings: pacman.Settings =
             pacman.Settings.instance();
6      conditions
7        enabled : settings.scoringEnabled();
8      inputfilters
9        scoreF : After = (enabled &
             selector=="gameStart")
10            {target=score;
                 selector="initScore"}
11         cor (enabled &
               selector=="eatFood")
12            {target=score;
                 selector="scoreFood"}
13         cor (enabled &
               selector=="eatVitamin")
14            {target=score;
                 selector="scoreVitamin"}
15         cor (enabled &
               selector=="eatGhost")
16            {target=score;
                 selector="scoreGhost"}
17         cor (enabled &
               selector=="nextLevel")
18            {target=score;
                 selector="scoreLevel"};
19   }
20
21   superimposition{
22     selectors
23       classes = { C |
             isClassWithNameInList(C,
             ['pacman.Game',
             'pacman.Level']) };
24     filtermodules
25       classes <- scoring;
26   }
27 }
```

Listing 12 Composition filters specification for `Scoring` concern

Filter Module Definition Lines 2 to 19 show the definition of the filtermodule `scoring`. Two external objects, `score` and `settings`, are referenced in the definition of the *externals* on Lines 3 to 5. Line 7 defines a *condition*, which is used in the filter specification. The filter module defines one *filter*, on Lines 9 to 18. The filter consists of several different parts, as indicated below:

$$\overbrace{scoreF}^{identifier} : \overbrace{After}^{filter\ type} = \overbrace{(enabled \,\&\, selector == \text{“}gameStart\text{”})}^{matching\ part} \overbrace{\{target = score; selector = \text{“}initScore\text{”}\}}^{substitution\ part}$$
$$cor \ \overbrace{(\ldots)\,\{\ldots\}}^{filter\ element}$$
$$\ldots$$

The *identifier* is the name of the filter in the filter module. The *filter type* specifies the type of the filter. In this example, the type is `After`, which means that an additional message is sent after the original message has been further processed. In this way behavior can be added after the original behavior. In the example, this behavior is to perform scoring. Examples of other filter types are `Dispatch`, which performs a dispatch of the message to a given target instead of the original target, and `Logging`, which performs logging of the given message.

Filters contain one or more *filter elements*. The filter `scoreF` contains five filter elements. Filter elements define message matching and substitution. The five filter elements in the example are composed with a *conditional-or* (`cor`) operator, meaning that if a filter element accepts, the filter accepts without evaluating the next filter elements. If a filter element rejects, the next filter element is processed.

A filter elements consists of a *matching part* and a *substition part*. The matching part defines a matching condition on the messages. Only if the matching condition is satisfied, the filter element accepts and the substitution part is executed. The substitution part changes certain properties of the message. When a filter element accepts a message, the filter of which the filter element is part accepts the message, and the behavior corresponding to the filter type is executed.

The example filter on Lines 9 to 18 of Listing 12 show five filter elements. Each of these filter elements only match when the condition `enabled` is `true`. Furthermore, each of these filter elements match a different selector (i.e., method call) and specifies a different selector in `score` to which a message is sent after the execution of the original message. For example, when a message with selector “`eatVitamin`” is processed, first `Level.eatVitamin` executes, followed by a call to the method `scoreVitamin` in `Score` (as shown on Lines 13 and 14), to apply the scoring that corresponds to eating a vitamin.

Superimposition Definition Lines 21 to 26 show the *superimposition* definition. A superimposition definition specifies which filter modules are placed (i.e., superimposed) on which artifacts (e.g., classes). The given superimposition definition places the filter module `scoring` on classes `Game` and `Level`.

References

1. The 20-sim tooling. http://www.20sim.com. Accessed April 2012
2. Web services business process execution language version 2.0. OASIS Standard (2007)
3. Abreu, R., Zoeteweij, P., van Gemund, A.: Spectrum-based multiple fault localization. In: 24th IEEE/ACM International Conference

on Automated Software Engineering, 2009. ASE '09, pp. 88–99. (2009). doi:10.1109/ASE.2009.25
4. Akşit, M., Bergmans, L., Vural, S.: An object-oriented language-database integration model: the composition-filters approach. In: Madsen, O.L. (ed.) Proceedings of the 7th European Conference on Object-Oriented Programming, pp. 372–395 (1992). http://trese.cs.utwente.nl/publications/paperinfo/LanguageDbase.pi.top.htm
5. Akşit, M., Tripathi, A.: Data abstraction mechanisms in sina/st. In: Proceedings of the Conference on Object-Oriented Systems, Languages and Applications. ACM Sigplan Notices, vol. 23, pp. 267–275 (1988)
6. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting object interactions using composition filters. In: Proceedings of the Workshop on Object-Based Distributed Programming. pp. 152–184. Springer, London (1994)
7. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: 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. pp. 345–364. ACM, New York, NY, USA (2005). doi:10.1145/1094811.1094839
8. Bapty, T., Neema, S., Scott, J., Sztipanovits, J., Asaad, S.: Model-integrated tools for the design of dynamically reconfigurable systems. Tech. Rep. ISIS-99-01, Institute for Software Integrated Systems, Vanderbilt University (1999)
9. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI. Lecture Notes in Computer Science, pp. 44–57. Springer, Berlin (2004)
10. Berbers, Y., Rigole, P., Vandewoude, Y., Baelen, S.V.: CoConES: CoConES: an approach for components and contracts in embedded systems. Lecture Notes in Computer Science, vol. 3778, pp. 209–231 (2005)
11. van den Berg, K., Conejero, J.: A conceptual formalization of crosscutting in aosd. In: Proceedings of the Desarrollo de Software Orientado a Aspectos (DSOA2005). Granada, Spain (2005)
12. Bergmans, L., Akşit, M.: Principles and design rationale of composition filters. In: Aspect-Oriented Software Development, pp. 63–95. Addison-Wesley, Boston (2005)
13. Bishop, R.H.: Modern Control Systems Analysis and Design Using MATLAB and SIMULINK. Addison Wesley, Boston (1996)
14. Bockisch, C.M.: An efficient and flexible implementation of aspect-oriented languages. Ph.D. thesis, Technische Universität Darmstadt, Germany (2008)
15. Bouraqadi, N., Ledoux, T.: Supporting AOP using reflection. In: Aspect-Oriented Software Development, pp. 261–282. Addison-Wesley, Boston (2005)
16. Broenink, J.: Modelling, simulation and analysis with 20-sim. Journal A **38**(3), 22–25 (1997)
17. Brooks, F.: No silver bullet essence and accidents of software engineering. Computer **20**(4), 10–19 (1987). doi:10.1109/MC.1987.1663532
18. Chen, F., Roşu, G.: 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. pp. 569–588. ACM, New York, NY, USA (2007). doi:10.1145/1297027.1297069
19. Delgado, N., Gates, A., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. **30**(12), 859–872 (2004). doi:10.1109/TSE.2004.91
20. van Deursen, A., Klint, P.: Little languages: little maintenance. J. Softw. Maint. **10**, 75–92 (1998)
21. van Dijk, W., Mordhorst, J.: CFIST. Composition Filters in Smalltalk. Graduation Report, HIO Enschede, The Netherlands (1995)
22. Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003). doi:10.1109/JPROC.2002.805829
23. Feldman, A., Provan, G., van Gemund, A.: The Lydia approach to combinational model-based diagnosis. In: Proceedings of the Twentieth International Workshop on Principles of Diagnosis (DX'09), Stockholm Sweden. pp. 403–408. Erik Frisk and Mattias Nyberg and Mattias Krysander and Jan slund (2009)
24. Filman, R.E., Elrad, T., Clarke, S., Akşit, M. (eds.): Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
25. Forbus, K.D.: Qualitative process theory. Artif. Intell. **24**(1–3), 85–168 (1984). doi:10.1016/0004-3702(84)90038-9
26. Francez, N., Hailpern, B., Taubenfeld, G.: Script: a communication abstraction mechanism and its verification. Sci. Comput. Program. **6**, 35–88 (1986)
27. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Boston (1995)
28. Glandrup, M.: Extending C++ using the concepts of composition filters. Master's thesis, University of Twente (1995). http://trese.cs.utwente.nl/publications/paperinfo/glandrup.thesis.pi.top.htm
29. Gray, J., Bapty, T., Neema, S., Schmidt, D.C., Gokhale, A., Natarajan, B.: An approach for supporting aspect-oriented domain modeling. In: Proceedings of the 2nd international Conference on Generative Programming and Component Engineering, GPCE '03, pp. 151–168. Springer, New York (2003)
30. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: specifying behavioral compositions in object-oriented systems. ACM SIGPLAN Notices **25**(10), 169–180 (1990)
31. Henzinger, T., Kirsch, C., Sanvido, M., Pree, W.: From control models to real-time code using giotto. Control Systems, IEEE **23**(1), 50–64 (2003). doi:10.1109/MCS.2003.1172829
32. Holland, I.M.: Specifying reusable components using contracts. In: Proceedings of the European Conference on Object-Oriented Programming. pp. 287–308. Springer-Verlag, London, UK (1992)
33. Kiczales, G., Rivieres, J.D.: The Art of the Metaobject Protocol. MIT Press, Cambridge (1991)
34. Kieburtz, R.B., McKinney, L., Bell, J.M., Hook, J., Kotov, A., Lewis, J., Oliva, D.P., Sheard, T., Smith, I., Walton, L.: A software engineering experiment in software component generation. In: Proceedings of the 18th International Conference on Software Engineering, ICSE '96, pp. 542–552. IEEE Computer Society, Washington, DC, USA (1996)
35. Kitchin, D., Quark, A., Cook, W.R., Misra, J.: The Orc programming language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) Proceedings of FMOODS/FORTE 2009. Lecture Notes in Computer Science, vol. 5522, pp. 1–25. Springer, Berlin (2009). doi:10.1007/978-3-642-02138-1-1
36. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. Artif. Intell. **32**(1), 97–130 (1987).doi:10.1016/0004-3702(87)90063-4
37. Kleijn, C.: 20-sim 4.1 Reference Manual (2009)
38. Koopman, P.: Embedded system design issues (the rest of the story). In: Proceedings of the 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1996. ICCD '96, pp. 310–317 (1996). doi:10.1109/ICCD.1996.563572
39. Maes, P.: Concepts and experiments in computational reflection. ACM SIGPLAN Notices **22**(12), 147–155 (1987)
40. Malakuti Khah Olun Abadi, S., Bockisch, C.M., Akşit, M.: 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, pp. 31–40. IEEE Computer Society Press (2009)
41. Markovski, J., van Beek, D., Theunissen, R., Jacobs, K., Rooda, J.: A state-based framework for supervisory control synthesis and veri-

fication. In: 49th IEEE Conference on Decision and Control (CDC), 2010. pp. 3481–3486 (2010). doi:10.1109/CDC.2010.5717095
42. Octopus project, ESI (2010). http://www.esi.nl/projects/octopus
43. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. In: Advances in Computers. pp. 329–400. Academic Press, London (1998).
44. Passerone, R., Damm, W., Ben Hafaiedh, I., Graf, S., Ferrari, A., Mangeruca, L., Benveniste, A., Josko, B., Peikenkamp, T., Cancila, D., Cuccuru, A., Gerard, S., Terrier, F., Sangiovanni-Vincentelli, A.: Metamodels in Europe: languages, tools, and applications. IEEE Des. Test Comput. **26**(3), 38–53 (2009)
45. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987). doi:10.1016/0004-3702(87)90062-2
46. de Roo, A.: Towards more robust advice: Message flow analysis for composition filters and its application. Master's thesis (2007). http://doc.utwente.nl/67050/
47. de Roo, A.: Managing software complexity of adaptive systems. Ph.D. thesis, Enschede (2012). http://doc.utwente.nl/79570/
48. de Roo, A., Hendriks, M., Havinga, W., Durr, P., Bergmans, L.: 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 (2008)
49. de Roo, A., Sözer, H., Akşit, M.: An architectural style for optimizing system qualities in adaptive embedded systems using multi-objective optimization. In: Joint Working IEEE/IFIP Conference on Software Architecture, 2009 and European Conference on Software Architecture, WICSA/ECSA 2009, pp. 349–352. Cambridge, UK (2009)
50. de Roo, A., Sözer, H., Akşit, M.: Runtime verification of domain-specific models of physical characteristics in control software. In: Proceedings of the 5th IEEE International Conference on Secure Software Integration and Reliability Improvement, Korea (2011)
51. Theunissen, R., Schiffelers, R., van Beek, D., Rooda, J.: Supervisory control synthesis for a patient support system. In: Proceedings of the European Control Conference (2009)
52. University of Twente: Compose* Annotated Reference Manual. http://composestar.sourceforge.net/content/annotated-reference-manual (2012)
53. University of Twente: COMPOSE*. http://composestar.sourceforge.net. Accessed April 2012
54. VDC Research: The embedded software and tools market intelligence service (2010)
55. Wichman, J.C.: The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, University of Twente (1999). http://trese.cs.utwente.nl/publications/paperinfo/wichman.thesis.pi.top.htm
56. Yoshioka, M., Umeda, Y., Takeda, H., Shimomura, Y., Nomaguchi, Y., Tomiyama, T.: Physical concept ontology for the knowledge intensive engineering framework. Advanced Engineering Informatics **18**(2), 95–113 (2004). doi:10.1016/j.aei.2004.09.004
57. Zoeteweij, P., Pietersma, J., Abreu, R., Feldman, A., van Gemund, A.: Automated fault diagnosis in embedded systems. In: Second International Conference on Secure System Integration and Reliability Improvement, 2008. SSIRI '08, pp. 103–110 (2008). doi:10.1109/SSIRI.2008.48

Author Biographies

Arjan de Roo received his M.Sc. degree in computer science from the University of Twente in the Netherlands in 2007. He received his Ph.D. degree in 2012 from the same University. Currently, he is an independent entrepreneur.

Hasan Sözer received his B.Sc. and M.Sc. degrees in computer engineering from Bilkent University, Turkey, in 2002 and 2004, respectively. He received his Ph.D. degree in 2009 from the University of Twente, The Netherlands. From 2002 until 2005, he worked as a software engineer at Aselsan Inc. in Turkey. From 2009 until 2011, he worked as a post-doctoral researcher at the University of Twente. He is currently an assistant professor at Özyeğin University.

Mehmet Akşit holds an M.Sc. degree from the Eindhoven University of Technology and a Ph.D. degree from the University of Twente. Currently, he is working as a full professor at the Department of Computer Science, University of Twente and affiliated with the institute Center for Telematics and Information Technology.