

MOO: An architectural framework for runtime optimization of multiple system objectives in embedded control software

Arjan de Roo^a, Hasan Sözer^{b,*}, Lodewijk Bergmans^a, Mehmet Akşit^a

^a University of Twente, Enschede, The Netherlands

^b Özyeğin University, İstanbul, Turkey

ARTICLE INFO

Article history:

Received 29 June 2012

Received in revised form 28 March 2013

Accepted 1 April 2013

Available online 15 April 2013

Keywords:

Architectural framework
Multi-objective optimization
Runtime adaptation
Embedded systems
Control software

ABSTRACT

Today's complex embedded systems function in varying operational conditions. The control software adapts several control variables to keep the operational state optimal with respect to multiple objectives. There exist well-known techniques for solving such optimization problems. However, current practice shows that the applied techniques, control variables, constraints and related design decisions are not documented as a part of the architecture description. Their implementation is implicit, tailored for specific characteristics of the embedded system, tightly integrated into and coupled with the control software, which hinders its reusability, analyzability and maintainability. This paper presents an architectural framework to design, document and realize multi-objective optimization in embedded control software. The framework comprises an architectural style together with its visual editor and domain-specific analysis tools, and a code generator. The code generator generates an optimizer module specific for the given architecture and it employs aspect-oriented software development techniques to seamlessly integrate this module into the control software. The effectiveness of the framework is validated in the context of an industrial case study from the printing systems domain.

© 2013 Elsevier Inc. All rights reserved.

1. Introduction

A current trend in embedded systems is toward adaptivity under varying circumstances (e.g., environmental conditions, user needs and input). For example, current high-end printing systems perform adaptive optimization of multiple *system objectives*, such as maximizing productivity and minimizing energy consumption. They apply trade-off decisions concerning several user needs, conflicting quality attributes and objectives. Adaptive optimization results in more competitive systems that leverage better customer satisfaction and cost effectiveness.

Adaptive and optimized behavior is achieved by adjusting certain *decision variables* in the system. Examples of decision variables are the speed of the system and the temperature setpoint of a heating device. The values of the decision variables are often subject to constraints. For example, the speed of the system is limited to a maximum speed and the amount of consumed power is limited to the amount of power available. The problem of balancing multiple system objectives by influencing a set of decision variables

that are subject to constraints is known as multi-objective optimization (MOO) (Keeney and Raiffa, 1976). In modern embedded systems, the set of strategies and techniques to facilitate MOO are implemented in software, as part of the control logic.

Embedded systems can comprise many different controllers implemented in several software modules, each controlling a part of the system. As a result, MOO have to be realized by manipulating and coordinating many controllers, scattered throughout the embedded control software. For example, power consumption and productivity of the system cannot be controlled by a specific controller; they are influenced by the behavior of the system as a whole. This manipulation and coordination of many controllers introduces additional structural complexity within the embedded control software.

There is a lack of systematic methods and techniques to design and implement MOO in embedded control software and to manage the resulting complexity. Usually the implemented control behavior is sufficient, but not optimal, as better optimizing solutions would be too complex to implement. When implemented, the applied optimization techniques, controlled variables, constraints and related design decisions are not documented as a part of the architecture description. They usually remain implicit, making it hard to communicate, analyze, maintain and reuse the embedded control software. Moreover, the lack of explicit representation of the decision variables and their interrelationships

* Corresponding author at: School of Engineering, Özyeğin University, Nişantepe Mah. Orman Sk. No. 13, Alemdağ – Çekmeköy 34794, İstanbul, Turkey. Tel.: +90 216 564 9383; fax: +90 216 564 9057.

E-mail address: hasan.sozer@ozyegin.edu.tr (H. Sözer).

leads to implicit dependencies among the software modules. As a result, any modification of the controlled system and/or the control software becomes error-prone.

Previously, we have proposed an architectural style (de et al., 2009), which introduces a graphical notation to document the software architecture from the MOO point of view. This style enables the documentation of decision variables, constraints and trade-offs at the architectural level. In this work, we have extended the style with domain-specific models. In particular, we use SIDOPS+ (Broenink, 1997) for documenting physical characteristics/models regarding the controlled system. Several such physical models are implemented in embedded control software. In digital printing systems for instance, physical models are used for estimating the heat exchange among components like the *toner belt* and the *paper path*. In practice, however, these models are implemented in a general-purpose programming language and tangled with the control software, just like the implementation of MOO. The new style, which we call as the MOO architectural style, is not based on just one language or notation. It makes use of multiple domain-specific modeling languages (DSMLs) together.

In this paper, we also introduce a complete framework for documenting, analyzing and realizing MOO in embedded control software. We have developed a toolchain consisting of visual editors, analysis tools, code generators and weavers. The framework generates optimizers based on the MOO architectural model. It also composes the generated code with both the physical models and the rest of the control software by means of aspect-oriented software development techniques. The core application is kept modular and independent from the optimization details. As such, our approach supports the reusability and maintainability of the MOO solutions and physical models, which are modularized and specified with separate DSMLs. Separate specification of MOO aspects, related physical models and the control logic also enables us to perform domain-specific analysis and verification.

We have illustrated the effectiveness of our framework in the context of an industrial case study from the printing systems domain. We have modularized the specification of the MOO solution and the implementation of the physical models for a part of the control software that is responsible for heat control on the paper path. These specifications are verified by the toolchain of the framework. Part of the control software is automatically generated based on the verified MOO architectural model and the related physical models. The generated code is automatically composed with the rest of the control software. We have compared the performance of the automatically generated control software with three other alternatives that use state-of-the-practice engineering solutions for optimization. We have seen that our approach can lead to an increase in productivity by 20%, and it can decrease the energy consumption by 10%. We have also seen that our approach reduces the deviation in print quality. In addition, the control software turns out to be more modularized, reusable and maintainable.

The remainder of this paper is organized as follows. The next section provides background information on multi-objective optimization. Section 3 introduces the industrial case study and puts the problem in context. Section 4 presents an overview of our approach. We explain the MOO architectural style and the toolchain of the MOO framework in Sections 5 and 6, respectively. In Section 7, we explain the experimental setup and present the results. Finally Section 8 provides the conclusions and discusses some future work directions.

2. Background: multi-objective optimization (MOO)

MOO is a mathematical problem in which the goal is to optimize multiple objectives. The objectives can be influenced by a set of

decision variables. The values of these decision variables are subject to constraints. A MOO problem is defined as follows (Collette and Siarry, 2003):

Definition 1 (MOO problem). A MOO problem can be represented by a tuple $\langle o(x), g(x), h(x) \rangle$, where

- $x \in \mathbb{R}^n$ represents the value of the n decision variables.
- $o(x) \in \mathbb{R}^n \rightarrow \mathbb{R}^k$ is a vector of k objective functions in the decision variables x that provide a valuation for the k objectives.
- $g(x) \in \mathbb{R}^n \rightarrow \mathbb{R}^l$ is a vector of l functions that describe inequality constraints. An inequality constraint means that x should be chosen in such a way that the outcome of each of the l functions in g is smaller than 0.
- $h(x) \in \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a vector of m functions that describe equality constraints. An equality constraint means that x should be chosen such that the outcome of each of the l functions in g is equal to 0.

The solution to this problem are the $x \in \mathbb{R}^n$ that minimize $o(x)$ under the constraints $\forall i \in [1 \dots l] g_i(x) \leq 0$ and $\forall j \in [1, \dots, m] h_j(x) = 0$.

The constraints limit the possible values for the decision variables to a subset of \mathbb{R}^n . This subset is called the *feasible decision space*. The solution to the optimization problem is selected from the feasible decision space. Each value x in the feasible decision space can be input to the objective functions $o(x)$. The set of all objective values that result from applying all values in the feasible decision space to the objective functions is called *objective space* or *criterion space*.

Optimizing a single objective is straightforward, as there is a one-dimensional objective space in which there is one point that has a smaller objective value than all the other points in the objective space. But when there are multiple objectives, there is usually not a single point in the objective space that has the minimal/optimal value for all objectives. In this case, one point in the objective space is said to *dominate* another point if it improves upon at least one of the objectives, without compromising on the other objectives. In a MOO problem, there might not be a single point that dominates all other points. Instead, there can be multiple points in the objective space that are not dominated by any other point in the objective space. These points are called *Pareto optimal points* (Pareto et al., 1896). For practical applications, such as in the control of embedded systems, a single result/solution of the MOO problem is necessary. Based on the relative importance of the objectives (e.g., based on customer needs), a single optimal value can be selected. A preference relationship on the objectives is usually specified in the form of a *trade-off function*, providing a scalar value for each point in the objective space, and as such providing a total ordering relationship among the Pareto optimal points.

We have employed existing theory on multi-objective optimization (Ehrgott and Gandibleux, 2002; Marler and Arora, 2004; Yoon and Hwang, 1995; Eckenrode, 1965; Hwang and Yoon, 1981; Geilen et al., 2007) to form a mathematical basis for our approach. Hence, we do not contribute, but rather utilize existing methods and techniques in this domain. Several (open-source) implementations of multi-objective optimization algorithms exist, such as in *Gnu linear programming kit* (2012) and *lp.solve* (2012). Matlab provides a function, called *fmincon*, to perform optimization of a single objective (Find minimum of constrained nonlinear multivariable function, 2013), which can also be utilized together with a trade-off function regarding multiple objectives. We have employed this function within the code that is generated by the MOO framework.

3. Industrial case study and problem statement

We have developed and evaluated our approach within the context of the Octopus project (Octopus project, 2012). In this section,

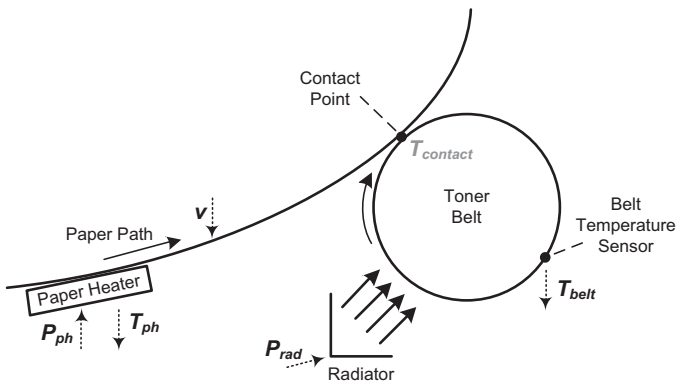


Fig. 1. Schematic view of the Warm Process.

we first describe the project context. Then, we introduce an industrial case study, taken from the digital document printing systems domain. Finally, we provide the problem statement.

3.1. The context

In traditional embedded systems development, trade-offs between (conflicting) system qualities (e.g., productivity, energy consumption) are made at design time. This results in embedded systems in which the trade-off between these qualities is fixed. For example, the embedded system is either a high-productive system with high energy consumption, or an energy-saving system with low productivity. Nowadays, market forces demand more flexible and adaptable machines, in which the trade-offs between system qualities can be made at runtime. For example, a customer of a printing system might sometimes need a high-productive machine (e.g., for time-critical print jobs), while in general an energy-saving system is preferred. The embedded system has to dynamically optimize the system qualities under changing circumstances, such as changes in environmental conditions and changes in user input. In the following subsection, we introduce an example case study in this context.

3.2. Case study: Warm Process

A part of the printing process in digital document printing systems is called the *Warm Process*. This process is responsible for transferring a *toner image* to paper.

Fig. 1 depicts a schematic overview of the parts in the printing system responsible for the Warm Process behavior. The Warm Process has two main parts; a *paper path* to transport sheets of paper and a *toner belt* to transport toner images. The *contact point* is the location where the paper path meets the toner belt. At this location the toner image is transferred from the toner belt to the sheet of paper. For correct printing, both the sheets of paper and the toner belt should have a certain temperature at the contact point. Therefore, the Warm Process contains two heating systems; a *paper heater* to heat the sheets of paper and a *radiator* to heat the toner belt. There is no sensor to measure the temperature of the toner belt at the contact point ($T_{contact}$). Instead, the physical system provides the following sensors and actuators:

- T_{ph} : Sensor that measures the paper heater temperature.
- T_{belt} : Sensor that measures the temperature at the sensor location on the toner belt.
- v : Actuator to set the printing speed.¹

- P_{ph} : Actuator to set the amount of power supplied to the paper heater.
- P_{rad} : Actuator to set the amount of power supplied to the radiator.

The printing speed can be adapted depending on the needs and objectives. For example, it can be lowered to reduce energy consumption or raised when printing on lighter paper (to increase productivity). If the speed of the system changes, the temperatures of the paper heater and belt also need to change, to maintain correct print quality. Engineers identified a relationship between the three variables: speed (v), paper heater temperature (T_{ph}) and the temperature of the belt at the contact point ($T_{contact}$). Correct print quality is ensured if this relationship, as follows, holds (c_1 , c_2 and c_3 are constants):

$$T_{contact}^{desired} = c_1 \cdot v - c_2 \cdot T_{ph} + c_3 \quad (1)$$

The paper heater reacts slowly to changing temperature setpoints, while the radiator can quickly influence $T_{contact}$. Therefore, engineers decided to mainly use the radiator to adjust the temperatures when the speed changes. This means that Eq. (1) is used to determine the required $T_{contact}$ (i.e., the setpoint). As there is no sensor to directly measure $T_{contact}$, engineers had to identify the following relationship between $T_{contact}$ and T_{belt} (c_4 is a constant):

$$T_{contact} = c_4 \cdot \frac{P_{rad}}{\sqrt{v}} + T_{belt} \quad (2)$$

In the following subsection, we analyze important aspects, design decisions and issues regarding the implementation of MOO in the context of the Warm Process case study.

3.2.1. Analysis of optimization in the Warm Process case study

In the Warm Process case study, engineers aim to introduce the possibility for the user to make trade-offs at runtime between the two conflicting objectives *power consumption* and *productivity* of the printing system. They identified the decision variables that can be used to influence the objectives, the different constraints in the system and the objective functions:

- **Decision variables:** Speed of the system (v), temperature setpoint of the paper heater (T_{ph}^{sp}).
- **Constraints:**
 - $60 \leq v$ (minimal speed is 60).
 - $v \leq 120$ (maximal speed is 120).
 - $40 \leq T_{ph}^{sp}$ (minimal setpoint for the paper heater is 40).
 - $T_{ph}^{sp} \leq 90$ (maximal setpoint for the paper heater is 90).
 - $P_{ph} \leq 1200$ (maximal power to the paper heater is 1200 W).
 - $P_{rad} \leq 800$ (maximal power to the radiator is 800 W).
 - $P_{total} \leq P_{avail}$ (total power consumption should not exceed the amount of power that is available to the system).
- **Objective functions:**
 - Total power consumption: $P_{total} = P_{ph} + P_{rad}$.
 - Productivity: $Prod = 1/v$ (productivity is defined as an inverse of speed, as in MOO the goal is to minimize the objective functions).

Fig. 2 shows the architecture of the control software for the Warm Process case study. The software architecture contains six

¹ Note that this only gives a simplified and abstract view of a printing system. In reality, there is no single actuator in the physical system to set the printing speed.

Instead, the paper path consists of multiple motors and pinches, each of which can be controlled independently. If this control is done in a coordinated way, this leads to correct paper transportation at a certain speed. Nevertheless, a virtual speed actuator can be implemented, which takes care of controlling the different motors in the paper path to obtain the requested speed.

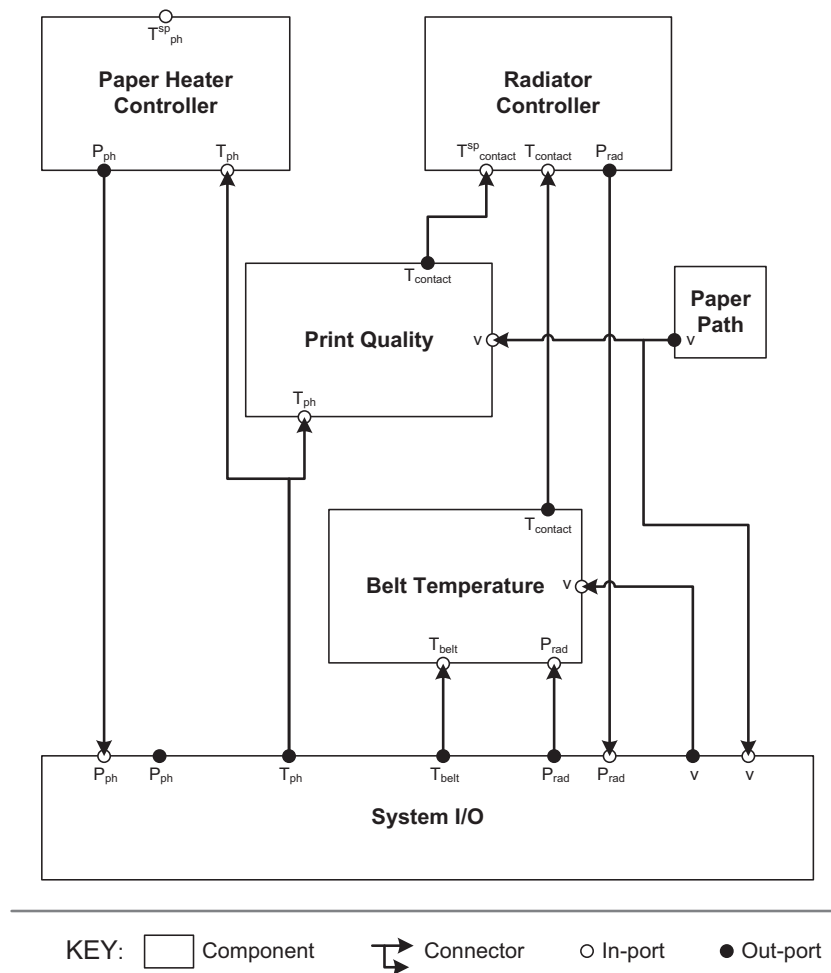


Fig. 2. Warm Process software architecture.

different components. Components have interfaces, which are represented as *in-ports/out-ports* to provide/retrieve a value to/from the component.

The *System I/O* component provides an interface to the sensors and actuators in the system. The component has an out-port for each sensor (T_{ph} and T_{belt}). The component has both an in-port and an out-port for each actuator (P_{ph} , P_{rad} and v). The out-port for an actuator provides the current level of the actuator (i.e., the last value that has been provided to the in-port).

PaperHeaterController and *RadiatorController* are components that contain the control logic for respectively the paper heater and the radiator. The components *PrintQuality* and *BeltTemperature* implement respectively the equation that determines print quality (Eq. (1)) and the equation that determines belt temperature (Eq. (2)). *PaperPath* is the component that controls the behavior of the paper path. It determines the speed of the system and provides this to the Warm Process control components.

If we relate the MOO-relevant variables to different ports in the software architecture, we can conclude that these variables are related to different architectural elements (e.g., ports), which are spread through the software architecture. This illustrates the fact that optimization can affect several components at different parts of the system. Depending on the scale of the system and the functionality being optimized, the scattering can be worse, which complicates the implementation and increases the maintenance effort.

3.3. Problem statement

There is a lack of systematic methods to design and implement MOO in embedded control software. This usually results in ad hoc solutions that are tailored to the specific system and therefore inflexible when the system changes or evolves. Moreover, the implementation of MOO gets tightly coupled and integrated with, and spread out over multiple control software components.² The overall impact is a reduction in software quality: ad hoc and tightly coupled solutions are more difficult to comprehend, their inflexible nature hinders their evolvability and reusability. The lack of a common methodology and corresponding terminology makes it harder to document and communicate the design decisions. The result is higher development and maintenance costs (Banker et al., 1993). Alternatively, as we have witnessed in practice, these problems can lead to the decision to remove the runtime optimization requirement, as the benefits of a more optimal system do not outweigh the reduced software quality and increased costs. Therefore, systematic methods, techniques and tools are required to support the realization of MOO and the management of software complexity introduced by its realization.

² The issue of crosscutting and a large number of dependencies was confirmed as a major issue by our industrial partners in the project.

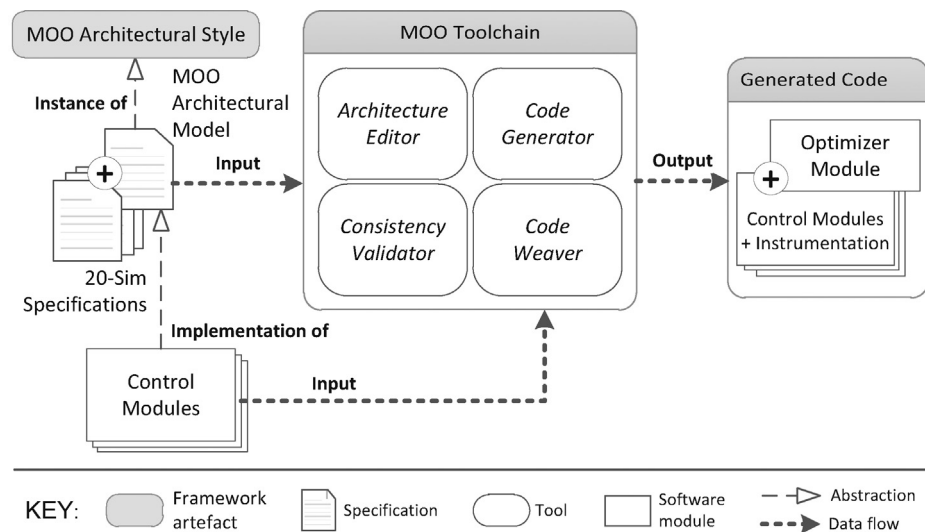


Fig. 3. The overview of the approach.

3.3.1. Related work

Optimization of software design has received more attention in recent years (Aleti et al., 2013). Usually there exist multiple (conflicting) quality attributes to consider in a software design (Sozer et al., 2011; Meedeniya et al., 2010). Therefore, MOO techniques have been applied to balance the feasible design alternatives with respect to different quality factors such as energy consumption vs. reliability (Aleti et al., 2009), resource utilization vs. data flow latency (Li et al., 2011), and performance vs. reliability (Sozer et al., 2011). In this paper we do not focus on the optimization of the design itself but the implemented control behavior instead. We aim at supporting the design and realization of a software system that employs MOO for controlling embedded systems.

An architectural framework (Calinescu et al., 2011) has been introduced for the development and adaptation of service-oriented systems. The framework employs MOO for selecting optimal composition of available services based on QoS requirements. In general, service-oriented systems are built through the dynamic composition of loosely coupled services. In this work, we focus on a different application domain that impose different challenges. In our problem domain, software structure is mainly fixed and it is embedded in a physical environment. The main challenge is to facilitate the optimal control of a set of devices that interact with this physical environment.

The implementation of multi-objective optimization in control systems has been previously discussed by Lui et al. (2002). We aim at introducing two new perspectives to these discussions. First, previous implementations mainly consider localized control problems, i.e., optimized behavior of a specific controller. Our goal is to optimize system qualities (e.g., energy consumption and the performance of the overall system), which has an impact on the system as a whole. Hence, many different controllers have to cooperate and an architectural focus is required to manage the incorporation of the MOO functionality as part of the control software. Second, previous discussions mainly focus on the mathematical design of an optimization algorithm. Our goal is to enable modular specification, analysis and automated composition of optimization techniques. Hence, in the following, we introduce the MOO framework to facilitate better management of software complexity in control software without compromising the effectiveness of optimization.

4. Overview of the MOO framework

We introduce the MOO framework to design and implement control software that employs MOO. The MOO framework employs an approach as depicted in Fig. 3 based on the *MOO architectural style* and the *MOO toolchain*.

The *MOO architectural style* provides the ability to specify a specialized Component-and-Connector model (Clements et al., 2002) for the architecture of the control software. The model includes elements of the MOO solution (decision variables, constraints and objective functions) in a structured way, as part of the architecture description. We refer to an architectural model that is created according to the MOO architectural style as *MOO architectural model* or *MOO model*.

In addition to the decision variables, constraints and objective functions, there exist computational logic and physical characteristics/models regarding the controlled system that influence the control behavior. To be able to analyze a MOO model and generate an optimizer module, these elements should also be specified. Therefore, the MOO method provides the possibility to refer to models that specify the computational logic (e.g., control logic, implemented physical characteristics) of components in the architecture. In this work, we adopt the SIDOPS+language of the 20-Sim toolset (Broenink, 1997; Kleijn, 2009), because of the suitability of this language to model control logic and physical characteristics. The models created with the SIDOPS+language are called 20-Sim models.

A MOO model together with the 20-Sim specifications are provided as input to the MOO toolchain. The toolchain contains a graphical editor, which is an extension of the ArchStudio 4 toolset (Dashofy et al., 2007), to create and edit MOO models. The *MOO Consistency Validator* checks the consistency of the MOO model. If the MOO model is consistent, it can be provided to the *MOO code generator*, to generate an optimizer module specific for the given architecture and given MOO solution. The software modules that implement the basic control architecture are provided to the *MOO code weaver*. The code weaver composes these software modules and the generated optimizer module by weaving instrumentation in the software modules. The result is embedded control software that includes MOO functionality. The next sections explain the MOO architectural style and the MOO toolchain in more detail.

5. MOO architectural style

In this section, we introduce the MOO architectural style³ for documenting embedded control software architecture from the multi-objective optimization point of view. Specific styles for control software have been developed before (Hofmeister et al., 2000), like the Process Control styles described in Shaw and Garlan (1996). However, these styles take a more general view on control; they describe the system in terms of a controlled system, sensors, controllers and actuators. The MOO style is applicable to a more specific type of functionality in control software, multi-objective optimization, and therefore can express specific properties of this functionality. In the following, we first provide a brief description of the MOO style, using the same description structure as used in Clements et al. (2002) to describe architectural styles. Then, the different characteristics of the style are described.



5.1. Style description

- **Elements:** Component;
- **Interfaces:** In-port, out-port;
- **Relations:** Connector;
- **Properties of elements:** *Component* implements the control logic, consisting of, among others, control algorithms and models of physical characteristics. Components have the following properties:
 - **20SimReference:** An optional reference to a 20-Sim model. This 20-Sim model describes the computational logic between the in-ports and out-ports of the component. If this computational logic is necessary to analyze the specified MOO solution, a reference to a 20-Sim model should be provided. Otherwise, it can be omitted.
 - **constraints:** A list of constraints on the variables corresponding to the ports of the component.
 - **isOblivious:** A flag that indicates whether the component is *oblivious*. This means that there is no software module that implements the component: the component is only used for specifying the MOO solution to be provided to the tooling.⁴ If the *isOblivious* flag is set, the component should always have a reference to a 20-Sim model.
 - **subModelReference:** An optional reference to another MOO model. This represents encapsulation of MOO models, to provide hierarchical application of the MOO style.
- **Properties of interfaces:** A port communicates the value of a specific (physical) variable. An in-port is used by a component to receive the value of a variable from other components. An out-port is used by a component to communicate the value of a variable to other components. In-ports and out-ports have the following properties:
 - **variableName:** String containing the name of the corresponding (physical) variable.
 - **constraints:** A list of constraints on the value of this port.
 - **isDecisionVariable:** A flag that indicates whether the variable corresponding to this port is a decision variable (i.e., the optimization algorithm may determine the value of the port).
 - **isObjective:** A flag that indicates that the variable corresponding to this port represents the outcome of one of the objective functions in the MOO problem.
- **Properties of relations:** Same as the C&C viewtype (Clements et al., 2002).

³ In the parlance of IEEE 1471 (Maier et al., 2001), the MOO architectural style can be regarded as a viewpoint.

⁴ Hence the term *oblivious* component, to indicate that the actual implementation of the software is not aware of this component.

Table 1
Notation of the MOO style.

Notation	Description
	Component with a <i>stereotype</i> and a <i>name</i> . Three stereotypes are available: <i>Analyzable</i> , <i>Oblivious</i> and <i>SubModel</i> . If a component has a reference to a 20-Sim model, then it has the stereotype <i>Analyzable</i> . If a component has the flag <i>isOblivious</i> set, then it has the stereotype <i>Oblivious</i> . Oblivious components have by definition a reference to a 20-Sim model, so the stereotype <i>Analyzable</i> is omitted. If the component references another MOO model (i.e., hierarchical composition), the stereotype is <i>SubModel</i> .
○	In-port
●	Out-port
◻■	In-port/out-port with the <i>isDecisionVariable</i> flag set.
▽	In-port/out-port with the <i>isObjective</i> flag set.
	Usage of a port: The ports that belong to a component are attached to the edge of the component. The port may be labeled with its <i>variableName</i> .
→	Connector
⊙	Informal label indicating that the component or port has constraints.

- **Topology:** Connectors connect ports. The *start-point* of a connector is always an out-port. The *end-point* of a connector is always an in-port. The semantics of a connector is that the value on the end-point of the connector (in-port) is set to the value of the start-point of the connector (out-port). An out-port can be the start-point of multiple connectors. An in-port cannot be the end-point of more than one connector. Each connector facilitates one-to-one connection. One can make use of multiple connectors to create multiple connections.

The MOO style has also a notation to create graphical representations of *MOO models*. These graphical representations are called *MOO views*. Table 1 shows the different elements of the notation and Fig. 4 shows the MOO view of the architecture for the Warm Process case study. The labels 1–5 indicate that constraints have been added to the corresponding port/component. This example MOO view will be used in the next subsections to illustrate the different characteristics of the style.

5.2. Specifying a MOO solution

This subsection explains the characteristics of the MOO style that can be used for specifying MOO concerns within a software architecture.

5.2.1. Ports providing decision variables and objective functions

Ports have the flags *isDecisionVariable* and *isObjective*. By setting the *isDecisionVariable* flag, a designer indicates that the port represents a decision variable. This means that the optimization algorithm may choose the value on the port. If the port also gets a value from a connector (in case of an in-port) or a component (in case of an out-port), the value provided by the optimization algorithm overrides the value provided by the connector or the component. Fig. 4 shows two ports with the *isDecisionVariable* flag set: The in-port T_{ph}^{sp} of the Paper Heater Controller component and the out-port v of the Paper Path component. This makes the speed and the setpoint of the paper heater the two decision variables in the MOO model.

By setting the *isObjective* flag, a designer indicates that the value on the port represents the outcome of an objective function. Fig. 4 shows two ports with the *isObjective* flag set: The out-port P_{total} of the Power component and the out-port *productivity*

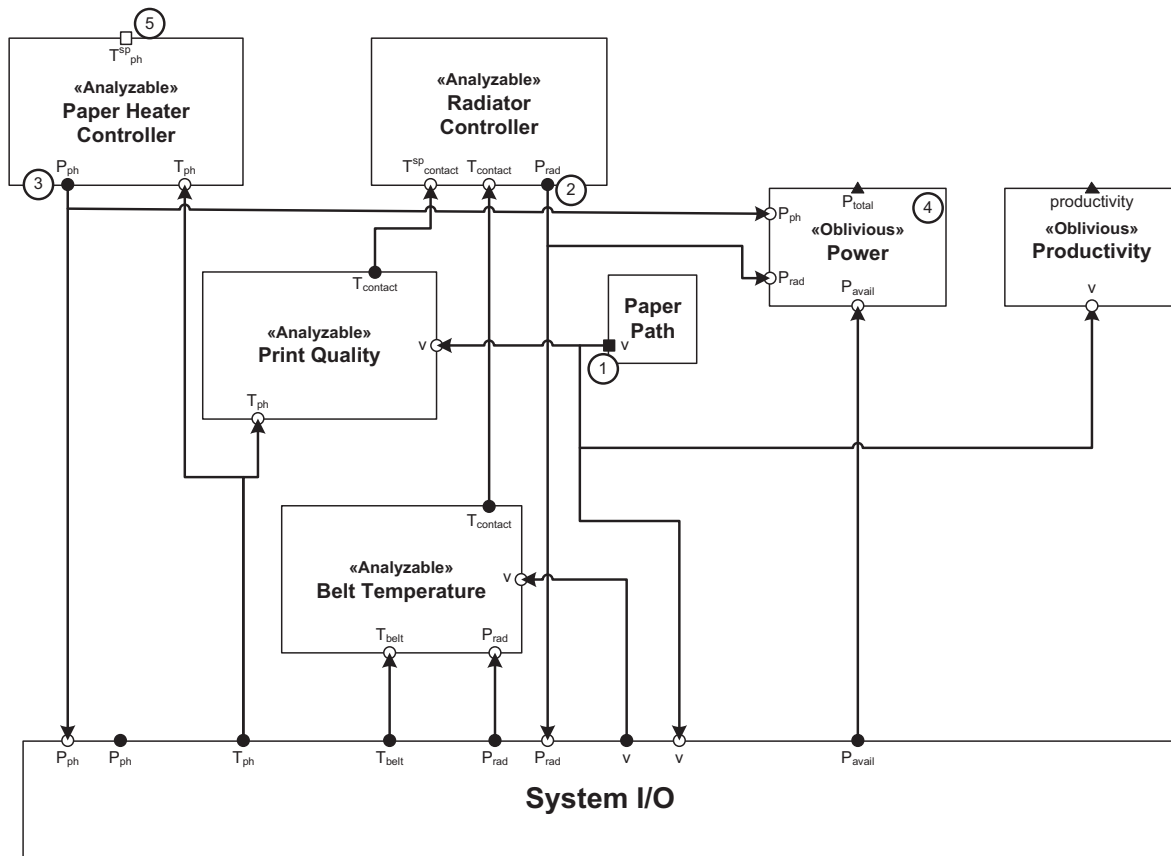


Fig. 4. MOO model of the Warm Process case study.

of the Productivity component. This means that there are two objectives in the system: total power consumption and productivity. Note that this MOO model does not specify a trade-off function between the two objectives.

5.2.2. Constraints

Components and ports have the property *constraints*, which contains a list of constraints. Component constraints provide a constraint among the values on the component's ports (e.g., $P_{total} \leq P_{avail}$). Port constraints only constrain the value on that specific port. As such, every constraint is associated with the related architectural element directly.

In the MOO view shown in Fig. 4, there are four ports (labeled 1, 2, 3 and 5) that have constraints and one component (labeled 4) that has constraints. These five constraints are:

- 1 Port: v
 - (a) $value \geq 60$
 - (b) $value \leq 120$
- 2 Port: P_{rad}
 - (a) $value \geq 0$
 - (b) $value \leq 800$
- 3 Port: P_{ph}
 - (a) $value \geq 0$
 - (b) $value \leq 1200$
- 4 Component: *Power*
 - (a) $P_{total} \geq 0$
 - (b) $P_{total} \leq P_{avail}$
- 5 Port: T_{ph}^{sp}
 - (a) $value \geq 50$
 - (b) $value \leq 90$

The two constraints on the out-port that is labeled 1 provide boundaries on the speed. The two constraints on the out-port labeled 2 provide boundaries on the power given to the radiator. Note that the optimizer is able to influence the value on this out-port by adjusting the speed; the control logic implemented within components *Print Quality* and *Radiator Controller* relate speed to the value on the out-port labeled 2, i.e., the power given to the radiator (P_{rad}). The two constraints on the out-port labeled 3 constrain the power given to the paper heater. It can be influenced by the decision variable T_{ph}^{sp} . The two constraints on the *Power* component limit the power consumption of the system to the amount of power available. The two constraints on the decision variable T_{ph}^{sp} give boundaries for this decision variable.

5.2.3. 20-Sim reference/analyzable component

In a MOO model, constraints and objective functions can be specified using variables (i.e., ports) other than the decision variables. However, there should be computational logic implemented in the components that provides mathematical relationships between the decision variables and the other variables used as part of constraints and objective functions. Otherwise, the constraints and objective functions cannot be influenced by the decision variables. To be able to analyze the MOO model, it should include these mathematical relationships. Therefore, the components of a MOO model have a property *20SimReference*. This property can be used to make a reference to a 20-Sim model that specifies the computational logic (e.g., a model of physical characteristics or continuous control logic) of the component. It is not necessary to include a 20-Sim model for each component. Such a model should be included only for components that relate constraints and objective functions to the decision variables.

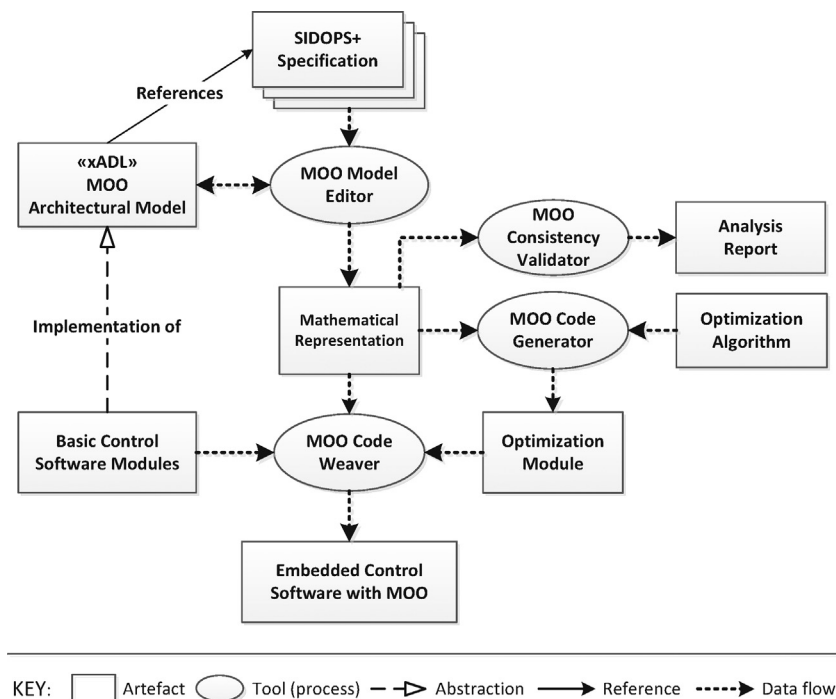


Fig. 5. Overview of the toolchain.

Fig. 4 shows four components with stereotype *Analyzable*. Their referenced 20-Sim models contain the physical relationships explained in Section 3.2.

Fig. 4 also shows two oblivious components: *Power* and *Productivity*. The purpose of these components is to model the objective functions and to model the constraint that the power consumption of the system is limited to the amount of power available. The *Power* component references a 20-Sim model containing the equation: $P_{total} = P_{ph} + P_{rad}$. The *Productivity* component references a 20-Sim model containing the equation: $productivity = 1/v$.⁵

In the following section, we introduce the MOO toolchain that can analyze a MOO model and generate code for realization of optimized control in embedded software.

6. MOO toolchain

Fig. 5 shows an overview of the MOO toolchain. The figure shows several artifacts and a number of (automated) processes that consume/generate certain (intermediate) artifacts. One of the artifacts is the MOO model that is the input to the toolchain. The MOO model can be edited by the *MOO Model Editor*. This editor is an extension of Archstudio (Dashofy et al., 2007). It provides a graphical modeling environment to create and edit MOO models. *MOO Model Editor* parses the stored MOO models and SIDOPS+specifications (containing 20-Sim models) to generate a *mathematical representation* of the MOO model, including the semantics from the referenced 20-Sim specifications. This mathematical representation is used by the *Consistency Validator* to check the consistency of the MOO solution. If the MOO solution is consistent, the *Code Generator* generates an *optimization software module*, based on a predefined/selected

optimization algorithm⁶ and the specific MOO problem provided by the mathematical representation. This optimization module needs to interact with the software modules that implement the basic control logic, to obtain values of certain (physical) variables and to influence the decision variables. The *Code Weaver* weaves this interaction into the control software modules. This results in embedded control software that includes MOO functionality. The different tools/processes/artifacts in the MOO toolchain are discussed in detail in the following subsections.

6.1. MOO Model Editor

The Archstudio toolsuite (Dashofy et al., 2007) offers a graphical editor for Component-and-Connector models. We extended this toolsuite to obtain a graphical editor for MOO models. Fig. 6 presents a screenshot of the extended Archstudio tooling, showing a *structural diagram* that represents the MOO model in Fig. 4.

Archstudio stores the architectural models in an architecture description language called xADL (Dashofy et al., 2001). xADL is an XML-based architecture description language that is defined using an XML schema. The xADL schema can be extended to be able to express new elements and properties in an architecture description. We extended xADL to be able to add the different elements of the realized MOO (decision variables, constraints and objective functions) to the components and ports in the structural diagrams of xADL. The extension is specified in the form of a XML schema (de Roo, 2012) conforming to the MOO architectural style.

6.2. Deriving a mathematical representation

This subsection describes the mathematical representation of a MOO model (together with the 20-sim models referred by its components), and explains how this representation is created. The basic

⁵ Strictly following the mathematical definition, the goal of MOO is to minimize the objective functions. Therefore, productivity is expressed as the inverse of speed: higher speed leads to a lower outcome of the productivity objective function.

⁶ The current toolset uses Matlab libraries (Find minimum of constrained nonlinear multivariable function, 2013) for optimization. In principle, any optimization algorithm/approach can be used.

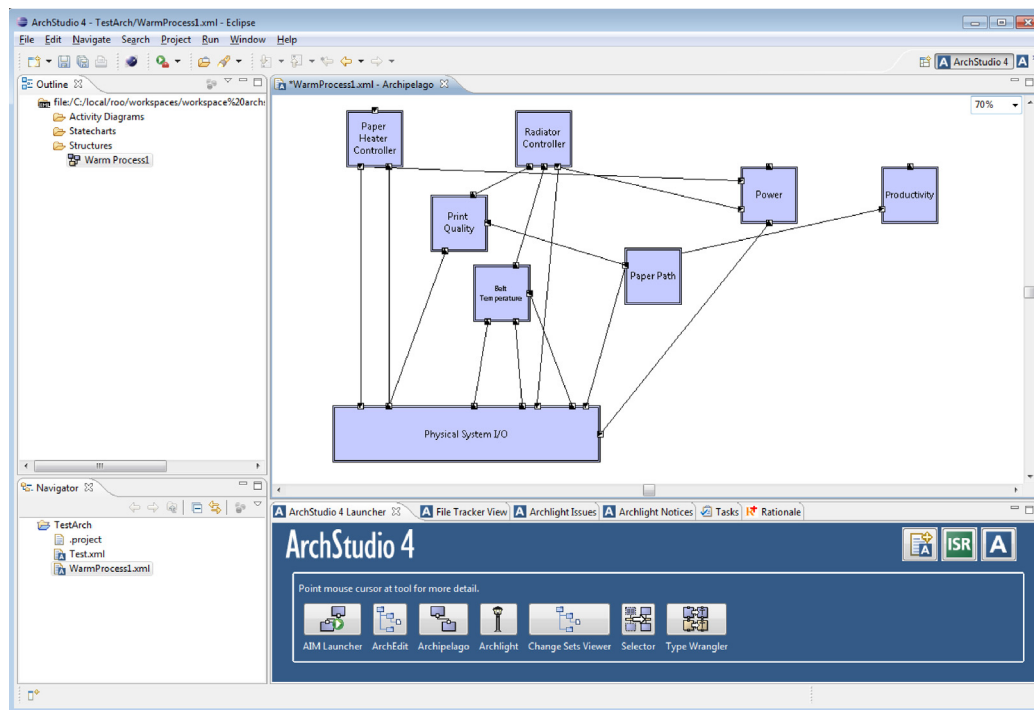


Fig. 6. Screenshot of the MOO Model Editor.

model of the representation is called a *derivation graph*. A derivation graph of a physical model is a directed graph that reflects how the values of the physical variables are derived from one another. A formal definition can be found in de Roo (2012). The derivation graph is created in two steps:

- 1 A derivation graph of each component in the MOO model is created.
- 2 The derivation graphs of the components are combined according to the connections between the components in the MOO model.

These two steps are explained in the following. For both steps, we apply standard data-flow analysis techniques.

Step 1. Creating a component's derivation graph. The derivation graph of a component is created using the following procedure:

Step 1a. Create the dependency graph. A *dependency graph* is a directed graph that relates variables and equations in a physical model to each other. A formal definition can be found in de Roo (2012). If the component references a SIDOPS+specification, this specification is parsed and the corresponding dependency graph is created (de Roo, 2012).

Fig. 7 shows an example component, two equations from the component's referred SIDOPS+specification and a constraint of the component. Fig. 8 shows the dependency graph that is created for this example.

Step 1b. Match ports to variable nodes Each port of a component corresponds to a variable. The variable nodes in a dependency graph

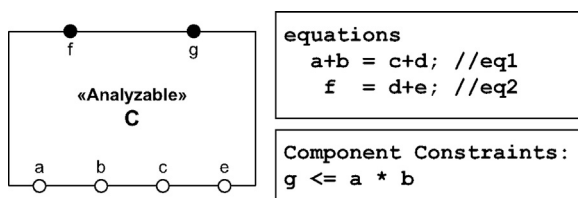


Fig. 7. Example component, SIDOPS+specification and component constraint.

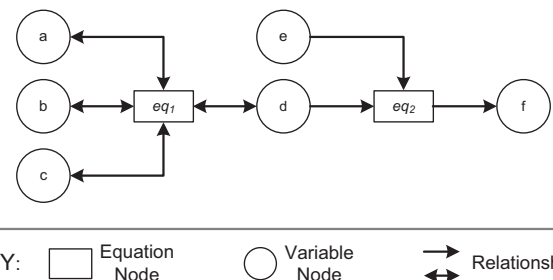


Fig. 8. Dependency graph corresponding to the example component and specification in Fig. 7.

also correspond to variables. Using name matching of the corresponding variables, ports of the component are matched to variable nodes in the dependency graph. For each port that has no matching variable node in the dependency graph, a new variable node is added to the dependency graph.⁷

The result of this step is the relationship *portMatching: ports* \rightarrow *vNodes*. This relationship is used by other processes in the MOO toolchain. Fig. 9 shows the matching of the component's ports to the variable nodes in the dependency graph. The out-port for variable *g* did not have a corresponding variable node. Therefore, a new variable node is added to the dependency graph.

Step 1c. Handling component's constraints. A component can have constraints attached.⁸ These constraints should be reflected in the component's dependency graph. To include a component's

⁷ This means that the variable corresponding to the port is not used in the SIDOPS+specification. Note that this does not mean that the component's implementation does not use (in case of in-ports) or compute (in case of out-ports) the values of that port. The SIDOPS+model of a component only needs to include the part of the semantics that relates constraints and objective functions to the decision variables. Therefore, the model may leave out unimportant relationships, in this way excluding certain variables.

⁸ Constraints can also be attached to ports. Constraints on ports only constrain the variable corresponding to that port, no other variables.

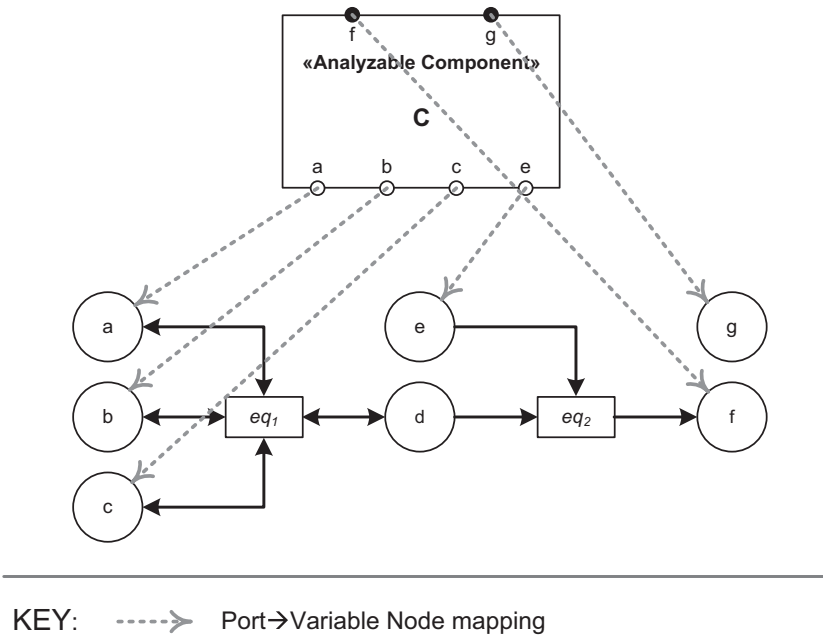


Fig. 9. Matching of ports to variable nodes.

constraint CC_i in the component's dependency graph, the following approach is adopted:

- 1 The constraint CC_i is rewritten in one of the following forms:
 - (a) $expressionCC_i < 0$, for 'greater than' and 'less than' constraints. For example, $g < a * b$ becomes $g - a * b < 0$.
 - (b) $expressionCC_i \leq 0$, for 'greater than or equal to' and 'less than and equal to' constraints. For example, $c \geq a + b$ becomes $a + b - c \leq 0$.
 - (c) $expressionCC_i = 0$, for equality constraints. For example, $b + c = d$ becomes $b + c - d = 0$.
- 2 A structure reflecting the mathematical equation $cc_i = expressionCC_i$ is added to the dependency graph. In this equation, cc_i is a new variable. $expressionCC_i$ is the expression formed in the previous step. The structure that is added to the dependency graph consists of an equation node, which reflects $expressionCC_i$ and a variable node, which reflects cc_i .
- 3 Depending on the type of the rewritten constraint in the first step of this approach, the cc_i variable node is annotated with the constraint $value < 0$, $value \leq 0$ or $value = 0$. The mapping from the constraint to the variable node is added to the relationship *constraintMatching*: $constraints \rightarrow vN_{arch}$. This relationship is used by other processes in the MOO toolchain.

Fig. 10 shows the component's dependency graph extended with an equation node CC_1 and a variable node cc_1 for the constraint. The constraint $g \leq a * b$ is rewritten as $g - a * b \leq 0$. The equation node CC_1 reflects the equation $cc_1 = g - a * b$. The cc_1 variable node gets annotated with the constraint $value \leq 0$.

Step 1d. Transform the dependency graph to the derivation graph. The dependency graph only specifies the dependencies between variables as specified by the 20-Sim model. It does not represent how the values of certain variables are derived from the values of other variables. For this purpose, a derivation graph is created (de Roo, 2012). The variable nodes that provide the input for the derivation are those variable nodes that match with the component's in-ports. Fig. 11 shows the component's derivation graph, which is created from the dependency graph in Fig. 10. The graph shows that the in-ports labeled a , b , c and e are independent variables for equations $eq1$ and $eq2$. Furthermore, the dependent variable d of

$eq1$ is an independent variable for $eq2$. The dependent variable of $eq2$ is f , which is an out-port of the component.

Step 2. Constructing the derivation graph for the entire MOO model. Suppose G is the set of derivation graphs of the components in a MOO model. The derivation graph $g_{arch} = (vN_{arch}, eN_{arch}, E_{arch})$ for the entire MOO model is then created by combining the derivation graphs in G , as follows:

- 1 The set of variable nodes in the derivation graph of the MOO model is the union of the sets of variable nodes in the derivation graph of each component: $vN_{arch} = \bigcup_{g_i=(vN_i, eN_i, E_i) \in G} vN_i$.
- 2 The set of equation nodes in the derivation graph of the MOO model is the union of the sets of equation nodes in the derivation graph of each component: $eN_{arch} = \bigcup_{g_i=(vN_i, eN_i, E_i) \in G} eN_i$.
- 3 The set of edges is constructed in two steps:
 - (a) First, create the union of the sets of edges in the derivation graph of each component: $E_{arch} = \bigcup_{g_i=(vN_i, eN_i, E_i) \in G} E_i$.
 - (b) For each connector in the architecture: suppose the corresponding out-port has matching variable node $n1 \in vN_{arch}$ and the corresponding in-port has matching variable node $n2 \in vN_{arch}$, then add $(n1, n2)$ to E_{arch} .⁹
- 4 The *portMatching* mapping for the MOO model is created by taking the union of the *portMatching* mappings for each component: $portMatching_{arch} = \bigcup_i portMatching_i$. The *portMatching* mapping is still needed to be able to relate nodes in the graph back to ports in the MOO model.
- 5 The *constraintMatching* mapping for the MOO model is created by taking the union of the *constraintMatching* mappings for each component: $constraintMatching_{arch} = \bigcup_i constraintMatching_i$.

Fig. 12 shows the derivation graph created for the example MOO model for the Warm Process case study. The derivation graphs created for the different components are indicated by dashed boxes.

⁹ Note that the in-port and out-port attached to a connector always have a corresponding variable node.

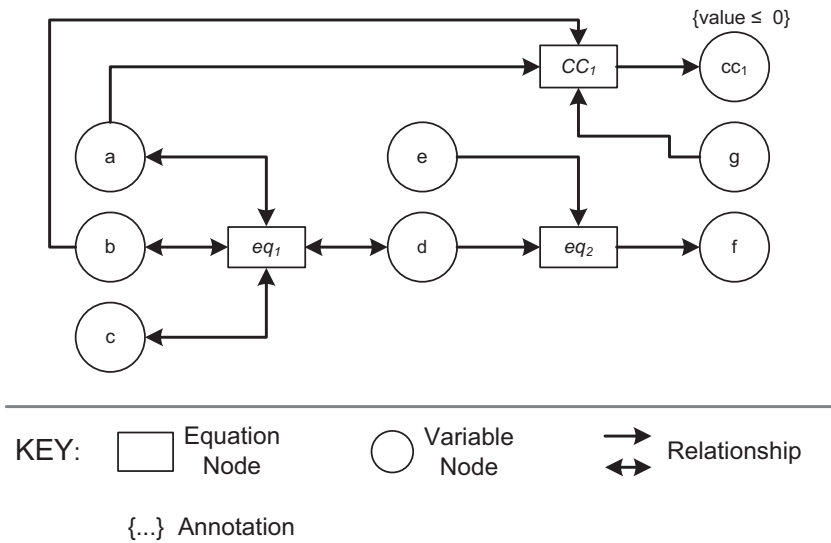


Fig. 10. Dependency graph extended with the component's constraint.

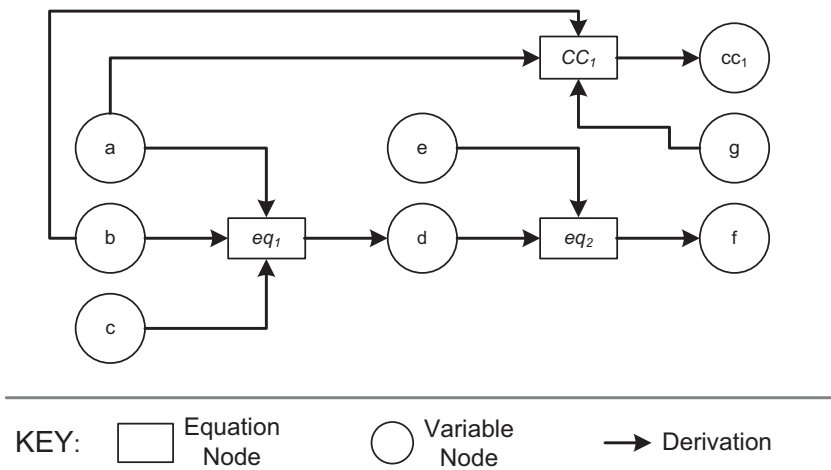


Fig. 11. The component's derivation graph.

6.3. MOO Consistency Validator

The MOO Consistency Validator analyzes a MOO model to ensure two properties: (i) whether all the constraints and objective functions are associated with a mathematical relationship in terms of the decision variables. (ii) if there exist components that need to have a reference to a SIDOPS+specification (i.e., 20-Sim model). The corresponding analysis steps are explained in the following.

6.3.1. Overall consistency analysis

If each constraint and objective function has a mathematical relationship with the decision variables, then an optimizer is able to influence all constraints and objective functions by changing the values of these variables. In this way, the optimizer can guarantee the satisfaction of the constraints and the optimization of the objective functions.¹⁰ The MOO Consistency Validator ensures this in two steps.

First, the MOO Consistency Validator performs a dependency analysis on the derivation graph to detect *dependent variable nodes*.

These nodes are associated with variables that are influenced by one or more decision variables. The analysis is performed by checking for each variable node whether there is a path to it from another variable node that corresponds to a decision variable. The algorithm is straightforward and explained in detail in de Roo (2012). Fig. 13 shows the set of dependent nodes among the variable nodes of the example derivation graph previously shown in Fig. 12. These are the nodes that are reachable from the decision variable nodes (v and T_{ph}^{sp}) in the graph.

In the second step, the MOO Consistency Validator checks for each node that has constraints or that is the outcome of an objective function, whether it is a dependent node or not. If not, this means that there is no mathematical relationship with the decision variables, so an inconsistency has been detected. Fig. 13 shows that all variable nodes that have constraints (the variable nodes labeled 1–5) are dependent variable nodes. Furthermore, the variable nodes that represent the outcome of objective functions (P_{total} and $Prod$) are dependent variable nodes. Therefore, the specified MOO model is consistent.

6.3.2. Detecting which components need a SIDOPS+specification

To construct a consistent MOO model, some of the components that take part in the architecture should have a reference to a

¹⁰ Under the assumption that the constraints themselves are consistent, i.e., feasible design space is not empty.

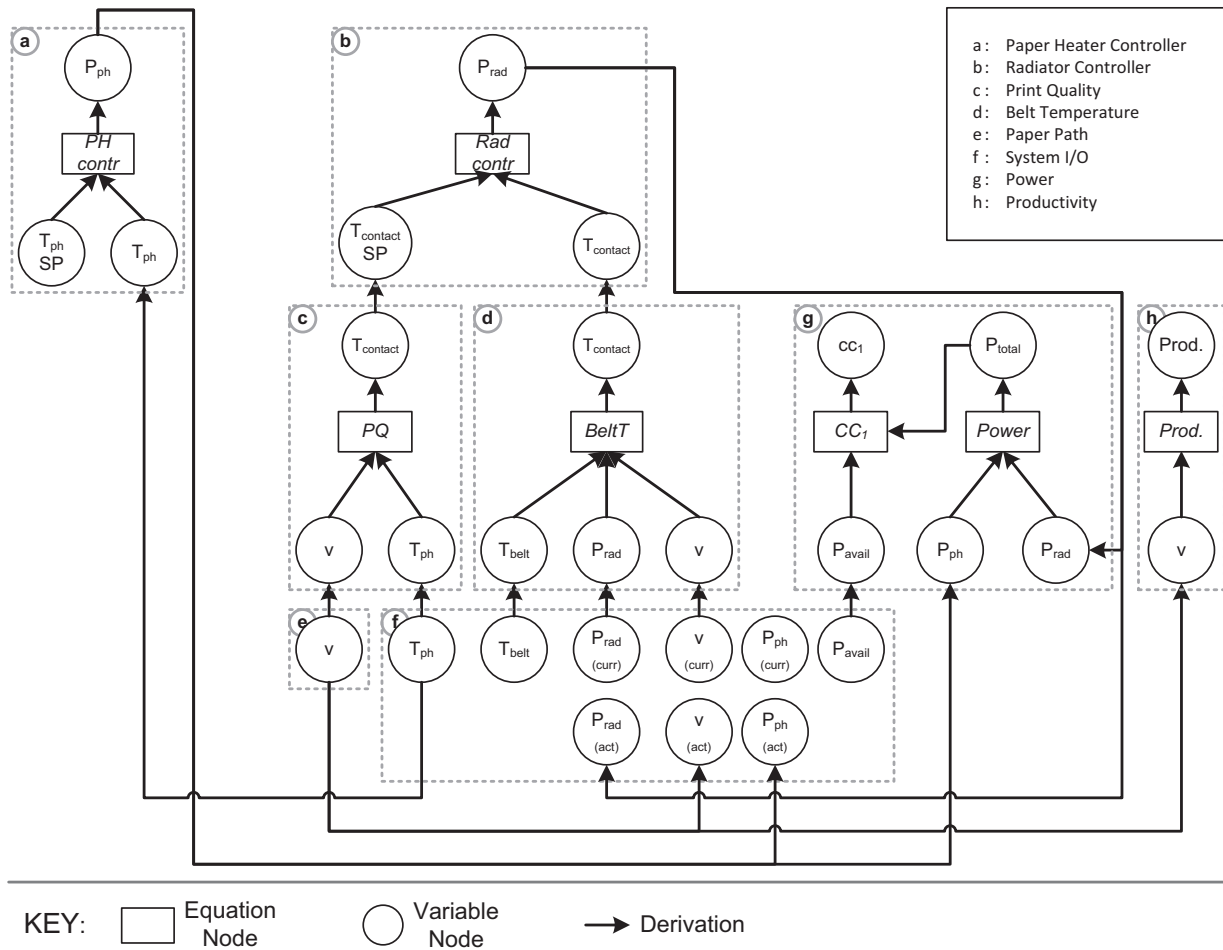


Fig. 12. Derivation graph for the example architecture.

SIDOPS+specification (i.e., 20-Sim model). The MOO Consistency Validator checks this property as follows.

First, a *structure graph* of the MOO model is created. A structure graph shows the potential relationships between ports as specified by the components and connectors. The algorithm for deriving the structure graph is straightforward and explained in detail in de Roo (2012).

Then, the following steps are performed to identify the need for a reference to a 20-Sim model for each port that has constraints or that is the outcome of an objective function, and for each component that has constraints:

- Find all paths in the structure graph from any decision variable node to the port/component node. Paths with cycles can be excluded, as they are not relevant for the result. Suppose the result is the set *paths*.
- There should be at least one $path \in paths$, for which all the components that correspond to the component nodes on this path refer to a 20-Sim model. Only in this case a mathematical relationship might be present that relates the constraints or objective function to the decision variables.

6.4. MOO code generator

MOO code generator is responsible for generating a software module that contains the optimization functionality specific for the given embedded control software. The following artifacts constitute as input to the tool:

- The MOO model: *mooModel*.
- The derivation graph of the MOO model.
- The *portMatching* relationship, which matches ports in the MOO model to variable nodes in the derivation graph.
- The *constraintMatching* relationship, which matches component constraints in the MOO model to variable nodes in the derivation graph.
- The set of dependent variable nodes.

These artifacts are first analyzed to extract the set of decision variables, a set of constraints on these decision variables and a set of objective functions regarding these decision variables. Detailed analysis procedures are provided in de Roo (2012). Next, code is generated based on the extracted information. The generated code performs the following tasks:

- 1 Information for certain parts of the MOO problem structure are collected from basic control components at runtime. This information can be the value at a certain port (`valueOf(aPort)`), a value in the state of a component (`valueInState(aVariable, aComponent)`), or a result of an expression in a 20-Sim model. The *MOO code weaver* instruments basic control components to facilitate access to the information.
- 2 Provide the MOO problem structure to an optimization algorithm (function). The optimization algorithm then determines a Pareto space or a single optimal outcome (in case there is a trade-off function). The current implementation only supports the Matlab function `fmincon` (Find minimum of constrained

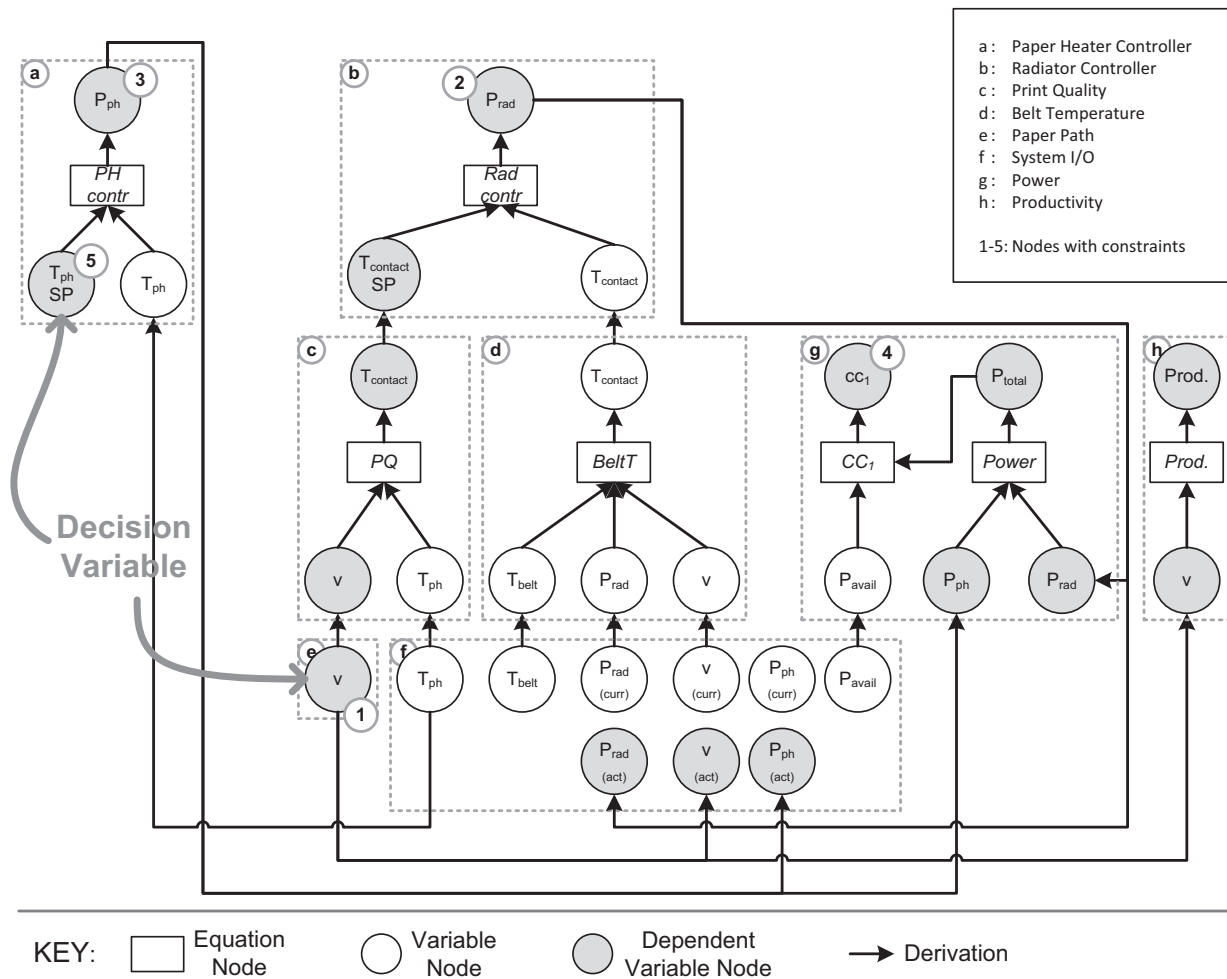


Fig. 13. Dependent nodes in the example derivation graph.

nonlinear multivariable function, 2013) as an optimization algorithm.

- 3 Set the decision variables in the basic control components to the result of the optimization.
- 4 Iterate over the above steps at a certain frequency (control loop).

6.5. MOO code weaver

The *MOO code weaver* tool instruments the basic control components. The optimization code generated by the *MOO code generator* uses this instrumentation to collect information from the basic control components and to set the optimized values for the decision variables. In this work, we assume that *get* functions are already available to obtain the required information. Aspect-oriented mechanisms are used to intercept calls to *set* functions of the decision variables and replace the argument with the value determined by the optimizer.

Previously, a layered architecture (Brunato and Battiti, 2010) has been proposed, where the implementation of the optimization techniques are modularized at the bottom layer, providing optimization functionality as a service to the upper layer(s). Even if the implementation of an optimization technique is modularized, it needs to access and update several variables scattered throughout the embedded control software. Therefore, a layered structure is not sufficient for proper separation of concerns since the necessary interactions of an optimizer crosscut the rest of the

control software. We employ specialized aspectual constructs for modularizing such crosscutting concerns (de Roo, 2012).

7. Experimentation and evaluation

We claim that the use of the MOO framework reduces development and maintenance effort, while being able to implement control software realizing multi-objective optimization. We make this claim for the following reasons.

First, our approach is automated end-to-end with a toolset. The manual effort is limited to the specification of the input models. Even most of these models (SIDOPS+specifications) are already being specified by domain engineers and directly reused in our approach.

Second, the use of the MOO architectural style makes MOO an explicit part of the architecture models. Decision variables, constraints, trade-offs and their dependencies are explicitly represented by ports, connectors and components of the software architecture. As such, software architecture design directly supports the realization and maintenance of a MOO implementation.

Third, our approach supports separation of concerns in a unique way. The control logic and application specific concerns are implemented in a general-purpose language (GPL). The physical phenomena is defined with a DSML. Our tools automatically compose (weave) artifacts defined in DSMLs with (into) software modules defined in GPLs. Hence, two different types of concerns

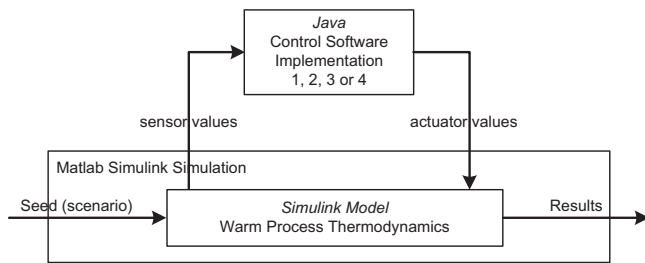


Fig. 14. Simulation setup.

are separated. In addition, they are implemented with different languages that are most appropriate for expressing the corresponding concerns.

We were also concerned with the impact of the MOO framework on quality attributes other than maintainability. In particular, custom implementations of state-of-the-practice engineering solutions might be considered better in terms of run-time performance. It turns out that the control software that is generated by the MOO framework actually performs even better than the manually implemented system. To evaluate our framework in this respect, this section describes an experiment based on the Warm Process case study. We have developed an embedded control software implementation using the MOO framework. This implementation is compared with three other state-of-the-practice engineering solutions with respect to the leveraged system *productivity*. We have used a Matlab Simulink model to simulate the printer hardware behavior and employed different simulation scenarios involving a limited and fluctuating amount of available power. In the following, we first discuss the experimental setup, followed by the implementation and results.

7.1. Experimental setup

We have developed an experimental setup based on a Matlab Simulink (MATLAB Simulink, 2012) model of the Warm Process thermodynamics. This model has been provided by our industrial partner, and is a realistic model of a real printer system. Fig. 14 shows the setup of this simulation. The simulation model can be seeded with a value for the pseudo-random number generator that determines the amount of power available in the Simulink model. After the simulation, the results concerning productivity can be obtained.

Besides the Simulink model of the Warm Process, the setup comprises one of the alternative control software implementations. The control software is implemented in Java and included in the Matlab environment, which natively supports a Java virtual machine. The following four control implementations are tested and compared¹¹:

- 1 *MOO*: An implementation created with the MOO framework.
- 2 *Intelligent Speed*: The *Intelligent Speed* algorithm (de Roo, 2012) is the state-of-the-practice approach, as we have learned from our industrial partner. This algorithm works as follows. The amount of power that is not utilized (P_{margin}) is calculated. When P_{margin} is too low, speed is decreased by 20 ppm. When P_{margin} is higher than a certain boundary (200 W), then the speed is increased by an amount that is proportional to the actual value of P_{margin} .
- 3 *Intelligent Speed 2*: This is a variant of the *Intelligent Speed* algorithm. The *Intelligent Speed* algorithm does not optimize the power margin P_{margin} to 0 W. Instead, it maintains a certain

amount of margin (varying between 0 W and 200 W) to cope with sudden drops in the amount of power available. In the experiment, such a margin results in lower productivity, as not all the available power is utilized. To make a fair comparison with respect to the MOO implementation, which optimizes for a power margin (P_{margin}) of 0 W, we also test a second, adapted implementation of the *Intelligent Speed* algorithm, called *Intelligent Speed 2* (de Roo, 2012). This algorithm adapts speed to optimize P_{margin} to approximately 0 W.

- 4 *Eco Mode*: The *Eco Mode* algorithm (de Roo, 2012) employs a strategy as follows; if the amount of available power is sufficient to print at the highest speed (120 ppm), then printing is performed at the highest speed. Otherwise, printing is performed at lowest possible speed (60 ppm).

The following settings and configurations are used with the experimental setup:

- $T_{contact}$ is maintained at the setpoint determined by the *PrintQuality* physical model (Eq. (1)).
- The speed of the system can vary between 60 and 120 ppm. It is assumed that the speed can be changed instantaneously.
- The paper heater temperature (T_{ph}) is controlled to its defined setpoint as long as the speed and the available power permit.
- 11 scenarios are tested with fixed amount of power available. Hereby, the amount of power available varies in fixed steps of 50 W from 700 W ($minP$) to 1200 W ($maxP$).
- 20 scenarios are tested with fluctuating power. Hereby, the amount of power available fluctuates between an amount sufficient to print at the highest speed ($maxP$) and an amount barely sufficient to print at lowest speed ($minP$). Each scenario is created using a pseudo-random generator which provides an even distribution between $minP$ and $maxP$. Experiments are performed for the following 7 different fluctuation intervals (i.e., the interval after which the amount of available power changes): 10 s, 25 s, 50 s, 100 s, 250 s, 500 s, and 1000 s.
- For each tested scenario, the simulation runs for 20,000 time steps (i.e., simulated seconds).

7.2. Experiment results

This section presents the results of the experiment. The four different implementations are compared according to the following four criteria:

- *Productivity*: The average speed of the system, in pages per minute (ppm).
- *Energy consumption*: The average energy consumed per printed page, in J/page.
- *Power margin*: The average power margin (the difference between the available and the consumed power), in W.
- *Print Quality*: The average deviation from perfect print quality, as defined by the *PrintQuality* physical model.

Figs. 15–18 show the results of the experiment for each of these four criteria. For each of the seven power fluctuation intervals the mean value over the 20 scenarios is shown.

Fig. 15 shows the results for the criterion *productivity*. Here we can see that for lower power fluctuation intervals, the MOO implementation performs significantly better than the other three control implementations. For the MOO, *Eco Mode* and *Intelligent Speed* implementation, the performance is stable for the different power fluctuation intervals. However, the *Intelligent Speed 2* implementation performs better with higher power fluctuation intervals, giving almost the same performance as the MOO

¹¹ Further details about these implementations can be found in de Roo (2012).

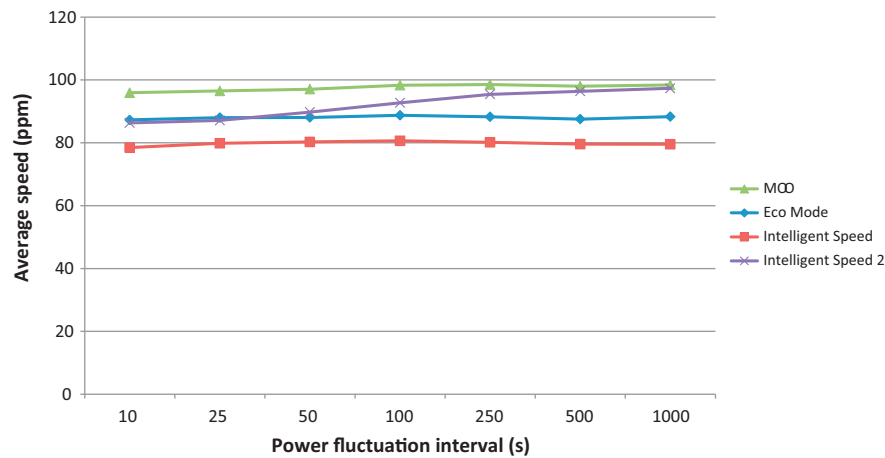


Fig. 15. Average printing speed.

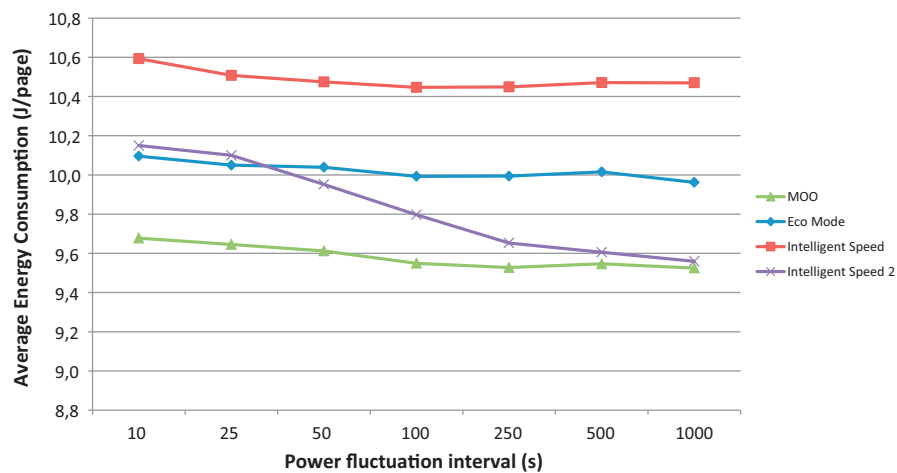


Fig. 16. Average energy consumption (lower \Rightarrow better).

implementation for the highest two power fluctuation intervals (500 s and 1000 s).

Fig. 16 shows the results for the criterion *energy consumption*. The figure shows that the MOO implementation performs significantly better than the Eco Mode and Intelligent Speed

implementation. The Intelligent Speed 2 implementation performs better with higher power fluctuation intervals.

Fig. 17 shows the results for the criterion *power margin*. As expected, the average power margin is high for the Intelligent Speed implementation. The Eco Mode implementation also shows

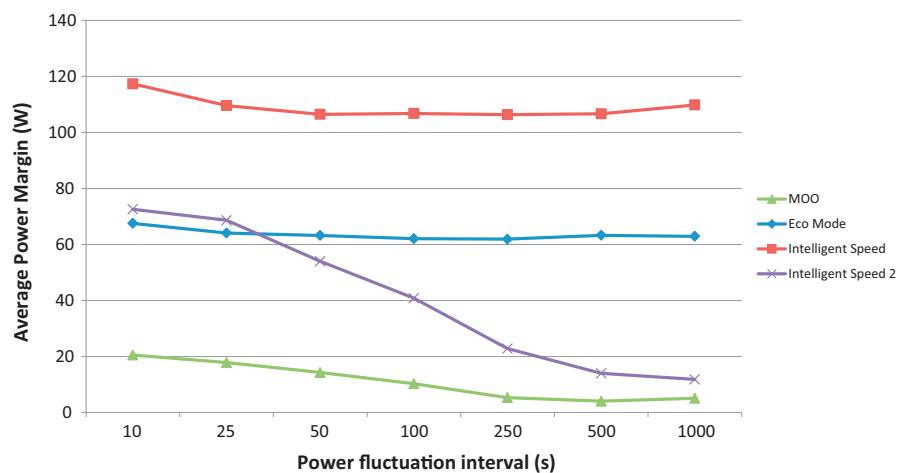


Fig. 17. Average power margin (lower \Rightarrow better).

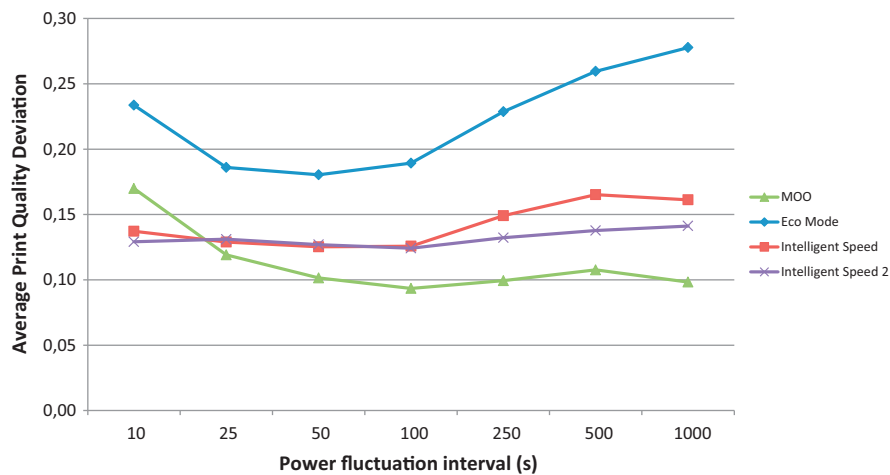


Fig. 18. Average deviation from print quality (lower \Rightarrow better).

a significant average power margin. The average power margin of the Intelligent Speed 2 implementation is lower for higher power fluctuation intervals. The MOO implementation has a low power margin. Note that the power margin of the MOO implementation is not necessarily 0, because for some scenarios, there is more power available than needed to print at the highest possible speed.

Fig. 18 shows the results for the criterion *print quality*. The figure shows that the Eco Mode implementation performs significantly worse than the other three implementations. The MOO implementation performs better than the other implementations, except for the lowest power fluctuation interval.

Fig. 19 shows the average speed (i.e., productivity) of the four implementations for the 11 scenarios with a constant amounts of power available within each scenario. The figure shows that with a constant amount of power available, the productivity of the Intelligent Speed 2 implementation is almost the same as the productivity of the MOO implementation. The Intelligent Speed and Eco Mode implementation perform less.

Fig. 20 shows the average speed (i.e., productivity) of the four implementations for the 20 test scenarios with a fluctuation interval of 1000. The figure shows that the MOO implementation performs consistently better than the Eco Mode and Intelligent Speed implementations. The performance of the Intelligent Speed 2 implementation is consistently almost the same as the performance of the MOO implementation.

7.3. Evaluation & discussion

In this subsection, we discuss experiment results and comment on a set of issues we have observed. First of all, Figs. 15 and 19 shows that MOO implementation has the best performance concerning productivity in all the tested scenarios. The difference with the Intelligent Speed and Eco Mode implementations is considerable, but this is mainly due to the fact that these implementations have large power margins. The Intelligent Speed 2 implementation approaches the performance of the MOO implementation for larger power fluctuation intervals, as it minimizes the power margin. For smaller power fluctuation intervals, the Intelligent Speed 2 implementation is not able to perform as good as the MOO implementation.

Also on the other three criteria, energy consumption, power margin and print quality, the MOO implementation performs equally well or better than the other three implementations. So, we can conclude that the MOO framework leads to systems that are able to function at the same or a higher level than systems applying other engineering solutions to optimize a system quality. Additionally, the MOO framework provides the ability to optimize multiple system qualities and dynamically make trade-offs between them. In this way, solutions created with the MOO framework differs from the other solutions, which typically optimize for a single system quality.

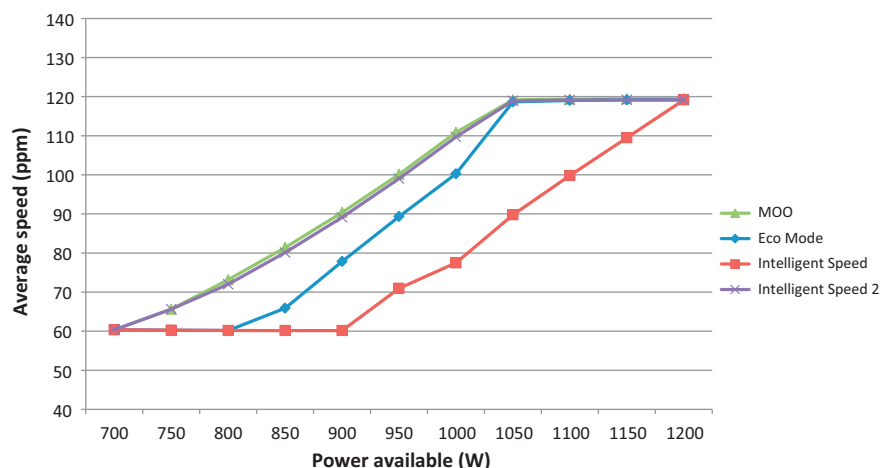


Fig. 19. Performance of the four implementations concerning productivity with a constant power supply.

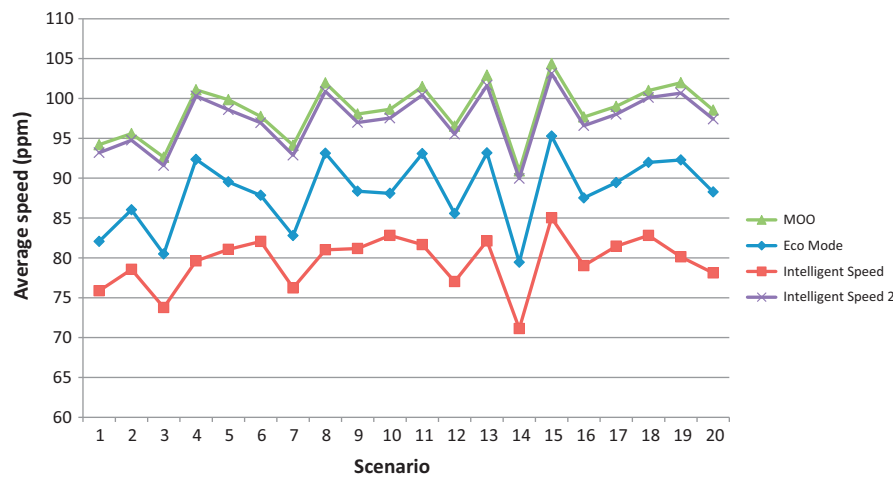


Fig. 20. Performance of the four implementations concerning productivity for 20 scenarios with a fluctuating power supply.

In addition, we observed that the control software created with the MOO framework is able to provide a precise and stable power margin, as opposed to Intelligent Speed algorithm, which gives an unpredictable margin between the amount of power available and the amount of power consumed (de Roo, 2012). We have also observed that the Eco Mode implementation has very instable speed behavior: the system accelerates and decelerates continuously. This behavior is undesirable from a user experience point of view and it increases wear-and-tear of the printer system. Also, the frequent speed changes lead to more deviation in print quality. Therefore, the Eco Mode implementation performs worst with respect to print quality, as can be seen in Fig. 18. We have observed smooth speed transitions for MOO, Intelligent Speed and Intelligent Speed 2 implementations. Experimental results and detailed discussions regarding these observations can be found in de Roo (2012).

Figs. 15 and 16 suggest that there is a reverse correlation between productivity and energy consumed per printed page. This reverse correlation can be explained from the fact that the printer loses an amount of energy to the environment which is unrelated to the speed of the system. If productivity is lower, the amount of energy lost to the environment is attributed to less printed pages, leading to a higher energy consumption per printed page.

8. Conclusion and future work

We have presented the MOO framework, to design and document MOO within the architecture of embedded control software. This framework has an architectural focus, based on the MOO architectural style. A MOO architectural model that confirms to this style defines essential elements of a MOO solution (i.e., decision variables, constraints and objective functions). The tools of the MOO framework can detect inconsistencies in the specification of the MOO solution. If the specification is consistent, it is used for generating the implementation of an optimizer. The tools also instrument the implementation of the control software components, to connect the generated optimizer to the control software components. This delivers an implementation of MOO in embedded control software.

We have applied the approach in the context of an industrial case study from the printing systems domain. Results showed that the control software developed with the MOO framework performs at least as good as, and typically better than state-of-the-practice solutions to optimize productivity. In addition, the MOO framework enables dynamic trade-offs between multiple objectives. Above all, the adoption of the MOO style

leverages better separation of concerns, modularity and maintainability. The style separates the implementation of the control logic and domain models that define physical phenomena. The control logic and the domain models are specified in different languages and can be maintained separately. The tools of the MOO framework automatically compose these artifacts.

Currently, the MOO architecture style captures only a static view (structure at runtime) of the optimization process. As future work, we are planning to extend the style with dynamic views. We also want to extend the style to include probabilistic distributions as constraints, as the relationships between decision variables might not be always exactly known.

Acknowledgements

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.

References

- Aleti, A., Bjørnander, S., Grunske, L., Meedeniya, I., 2009. ArcheOpterix: an extendable tool for architecture optimization of AADL models. In: *Proceedings of the Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES)*, pp. 61–71.
- Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., Meedeniya, I., 2013. Software architecture optimization methods: a systematic literature review. *IEEE Transactions on Software Engineering*, <http://dx.doi.org/10.1109/TSE.2012.64>.
- Banker, R.D., Datar, S.M., Kemerer, C.F., Zweig, D., 1993. Software complexity and maintenance costs. *Communications of the ACM* 36, 81–94.
- Broenink, J., 1997. Modelling, simulation and analysis with 20-sim. *Journal A* 38 (3), 22–25.
- Brunato, M., Battiti, R., 2010. Grapheur: a software architecture for reactive and interactive optimization. In: *Proceedings of the 4th International Conference on Learning and Intelligent Optimization*. Springer-Verlag, Berlin, Heidelberg, pp. 232–246.
- Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G., 2011. Dynamic QoS management and optimization in service-based systems. *IEEE Transactions on Software Engineering* 37 (3), 387–409.
- Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., Little, R., 2002. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston.
- Collette, Y., Siarry, P., 2003. *Multiobjective Optimization: Principles and Case Studies*, 1st ed. Springer-Verlag, Berlin.
- Dashofy, E.M., Hoek, A.V.d., Taylor, R.N., 2001. A highly-extensible, xml-based architecture description language. In: *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, IEEE Computer Society, Washington, DC, USA, p. 103. <http://dx.doi.org/10.1109/WICSA.2001.948416>.
- Dashofy, E., Asuncion, H., Hendrickson, S., Suryanarayana, G., Georgas, J., Taylor, R., 2007. Archstudio 4: an architecture-based meta-modeling environment. In: *ICSE COMPANION '07: Companion to the Proceedings of the 29th International Conference on Software Engineering*, IEEE Computer Society, Washington, DC, USA, pp. 67–68. <http://dx.doi.org/10.1109/ICSECOMPANION.2007.21>.

- de, A.R., Sözer, H., sit, M.A., 2009. 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 & European Conference on Software Architecture, WICSA/ECSA 2009, Cambridge, UK, pp. 349–352.
- de Roo, A. 2012. Managing software complexity of adaptive systems. Ph.D. Thesis. University of Twente.
- Eckenrode, R.T., 1965. Weighting multiple criteria. *Management Science* 2 (3), 180–192.
- Ehrgott, M., Gandibleux, X., 2002. *Multiple Criteria Optimization: State of the Art Annotated Bibliographic Surveys*. Kluwer Academic Publishing, Norwell, Massachusetts.
- Find minimum of constrained nonlinear multivariable function, 2013. <http://www.mathworks.nl/help/toolbox/optim/ug/fmincon.html> (accessed 2013).
- Geilen, M., Basten, T., Theelen, B., Otten, R., 2007. An algebra of pareto points. *Fundamenta Informaticae* 78 (1), 35–74.
- Gnu linear programming kit, 2012. <http://www.gnu.org/software/glpk/> (accessed 2012).
- Hofmeister, C., Nord, R., Soni, D., 2000. *Applied Software Architecture*, 1st ed. Addison-Wesley Professional, Reading, Massachusetts.
- Hwang, C., Yoon, K., 1981. *Multiple Attribute Decision Making: Methods and Applications: A State-of-the-Art Survey*, vol. 13. Springer-Verlag, Berlin.
- Keeney, R.L., Raiffa, H., 1976. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley & Sons, Inc., New York.
- Kleijn, C., 2009. 20-Sim 4.1 Reference Manual.
- Li, R., Etemaadi, R., Emmerich, M.M., Chaudron, M.V., 2011. An evolutionary multiobjective optimization approach to component-based software architecture design. In: *IEEE Congress on Evolutionary Computation*, pp. 432–439.
- lp_solve, 2012. <http://lpsolve.sourceforge.net/> (accessed 2012).
- Lui, G., Yang, J., Whidborne, J., 2002. *Multiobjective Optimisation and Control*. Research Studies Press, Baldock, Hertfordshire, England.
- Maier, M.W., Emery, D., Hilliard, R., 2001. Software architecture: introducing IEEE standard 1471. *IEEE Computer* 34 (4), 107–109.
- Marler, R., Arora, J., 2004. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization* 26, 369–395.
- MATLAB Simulink, 2012. <http://www.mathworks.com/products/simulink/> (accessed 2012).
- Meedeniya, I., Buhnova, B., Aleti, A., Grunske, L., 2010. Architecture-driven reliability and energy optimization for complex embedded systems. In: *Proceedings of the 6th International Conference on Quality of Software Architectures*. Springer-Verlag, Berlin, Heidelberg, pp. p52–67.
- Octopus project, 2012. ESI. <http://www.esi.nl/projects/octopus>
- Pareto, V., 1896. *Cours D'Économie Politique*. F. Rouge, Lausanne, Switzerland.
- Shaw, M., Garlan, D., 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- Sozer, H., Tekinerdogan, B., Aksit, M., 2011. Optimizing decomposition of software architecture for local recovery. *Software Quality Journal*, 1–38.
- Yoon, K., Hwang, C., 1995. *Multiple Attribute Decision Making: An Introduction*. Sage Publications, Thousand Oaks, California.
- Arjan de Roo** received his MSc degree in computer science from the University of Twente in the Netherlands in 2007. He received his PhD degree in 2012 from the same University. Currently, he is an independent entrepreneur.
- Hasan Sözer** received his BSc and MSc degrees in computer engineering from Bilkent University, Turkey, in 2002 and 2004, respectively. He received his PhD 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.
- Lodewijk Bergmans** is a part-time Assistant Professor at the Computer Science Department of the University of Twente in The Netherlands. He holds an MSc and PhD degree from the same institute. He has ample industrial experience in object-oriented software engineering, most notably on analysis and design, and software architecture of embedded systems. Lodewijk's long-term research goal is to achieve a deeper understanding of software composition. In addition, Lodewijk works at the Software Improvement Group, where he helps organisations making sound IT decisions, through a fact-based and metrics-based analysis of their software systems and development processes.
- Mehmet Akşit** holds an MSc degree from the Eindhoven University of Technology and a PhD 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 Centre for Telematics and Information Technology.