Model-based computing:
Developing flexible machine control software

Markus P.J. Fromherz a,*, Vijay A. Saraswat b,1, Daniel G. Bobrow a,2

a Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA  
b AT&T Labs, 180 Park Avenue, Florham Park, NJ 07932, USA

Received 15 January 1999

Abstract

In the conventional approach to simulating, controlling, and diagnosing a real-world physical system, engineers typically analyze the interactions of the system’s components and processes, and then develop new and dedicated code for that system. Instead, building on principles from model-based reasoning and constraint programming research, we propose an integrated approach to software development we call model-based computing. We present this approach in the context of control software for modular electro-mechanical systems. Our approach is used in commercial systems and has been shown to both simplify the development of machine control software, and make the software and the controlled systems more flexible and effective.

In this paper, building on a generic control software architecture, we first develop a domain theory with corresponding modeling language. Models capture a system’s capabilities from first principles and independently of the control task. We then introduce modeling technology using concurrent constraint programming, which gives our modeling approach a sound and powerful theoretical foundation. Constraint programming also brings with it a host of generic reasoning techniques such as deduction, abduction, and search, and we show how such techniques can be applied to the model-based configuration and control of our systems. We end with a review of how model-based computing can be extended to other tasks such as design and testing. We believe that together, models, task architecture, and reasoners offer a compelling framework for building software for computationally controlled systems. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Model-based reasoning; Modeling; Constraint programming; Control; Scheduling
1. Introduction

This paper describes an approach to developing real-time system-level controllers for electro-mechanical machines, based on the ideas of model-based reasoning: task-independent models of a system’s structure and behavior are used by system-independent algorithms to reason about the modeled system. In our approach, which we call model-based computing, system models become an integral and executable part of the system software, enabling the software to adapt itself to different configurations, and to flexibly react to changes in the system’s capabilities. Using our approach, the software development effort consists of three concurrent activities (Fig. 1):

1. the development of application-independent, declarative, constraint-based models of physical machine components and configurations,
2. the development or re-use of a separate, configuration-independent software architecture for the task at hand, and
3. the development or re-use of mediating reasoners that provide the glue for embedding the models into the task architecture.

An early version of our approach has been used in a family of Xerox mid-size printing products. The version described in this paper is being used on a routine basis in the development of a Xerox high-end printing product, and its modeling language and control software components have become part of a reusable machine control toolkit.

This approach is part of the larger vision for model-based computing: to support both human communication and computer processing through formal representations. Our goal is not only to increase the productivity of software developers, but also to improve the communication among different subsystem engineers (mechanics, electronics, software, etc.), to ensure the consistency across different engineering tasks (design, control, testing, diagnosis), and to enable automatic, modular configuration of the resulting systems. Formal but readable documents are the core of this vision.

Model-based computing relies on the use of domain-specific constraints for modeling, an idea that is very familiar to engineers working in a particular domain. However, it can be difficult to provide a simple semantic interpretation to a modeling language with domain-specific constraint systems, precluding the use of powerful tools to reason about the constraint representations. We therefore define our modeling language at two levels: a domain-specific, engineering-oriented modeling language, CDL (Component Description Language), and a domain-independent programming language based on concurrent constraint programming (CCP) [41]. Our approach is to translate (reversibly)
the higher-level modeling language into the lower-level CCP framework, and to apply all reasoning at the CCP level. CCP provides many of the desired characteristics for such an approach, including a logic interpretation and compositional construction. CCP offers an elegant, extensible, and customizable framework for declarative representation that supports a powerful concurrent computational interpretation. This dual interpretation allows the application of powerful static analysis techniques from the area of programming languages to manipulate and reason about models.

In this paper, we demonstrate our approach in the concrete context of controlling complex reprographic systems. First, we analyze the system control task and develop a domain theory for modeling the system’s capabilities (Section 2). Then, we present a corresponding modeling language and framework (Section 3), and outline reasoners that enable a model-based implementation of the controller (Section 4). Finally, we discuss a system-control approach based on our techniques (Section 5). We review related work in Section 6. While we focus on our experience with system-level control, we believe that these ideas are of much wider applicability, and we also review related work reported elsewhere, including simulation, productivity analysis, and design optimization (Section 7).

2. System control in modern reprographic machines

2.1. Reprographic machines

Modern digital reprographic systems come in many forms, from low-end scanners and printers to departmental multi-function devices, to high-end systems that run at well over 100 pages per minute. Reprographic machines consist of a source of paper and images, a complex paper path that brings these together at the right time, place and orientation, and finishing components that collate, sort, staple and bind the resulting, marked sheets. The paper path is a crucial element of this structure and is considered one of Xerox’s core competencies. We focus as our primary example on showing how model-based computing enables flexible generation of control code for moving paper along this path. 3

Large machines are typically split into modules such as “feeder”, “mark engine” and “finisher”. Feeders, housing several sheet trays, serve as sheet sources. The mark engine module processes and transfers images onto sheets. Finishers sort sheets, collect them in bins, and staple or bind them. High-end configurations typically consist of multiple feeder and finisher modules, connected in series and with a mark engine module in between.

Fig. 2 shows a typical configuration of a mid-size print engine with three modules: a feeder module with three feed trays, a mark engine module that is able to produce single and double-sided prints (simplex and duplex sheets), and a finisher with two output trays.

Modules themselves consist of multiple components. The mark engine module (Fig. 3), for example, consists of a photo-receptor belt, an image transfer component, a sheet inverter, path merging and splitting components, and two simple sheet transport components

3 All examples of machine configurations and used scenarios are realistic but usually simplified, and none should be taken as describing an existing or future Xerox product.
(registration and duplex loop). The transfer component prints an image onto one side of the sheet from a continuously revolving photo-receptor belt onto which images are laid in the form of charged toner particles. The inverter has two modes of operation (Fig. 4); as it will be our running example in this paper, we explain it in more detail. In one mode of operation, the sheet is guided by the inversion gate $G$ from the input rollers $R_{\text{in}}$ down into the inversion rollers $R_{\text{inv}}$; when its trailing edge clears the gate, the sheet is stopped and then moved in reverse direction up and through the output rollers $R_{\text{out}}$. In the other mode, the sheet is moved from the input rollers, guided by the inversion gate in its up position, directly to the output rollers. Thus, the first mode inverts a sheet from face-up to face-down orientation or vice versa, while the second mode bypasses this operation and leaves the sheet’s orientation unchanged. The other components of the mark engine module primarily move sheets along the paper path.

In this configuration, a *simplex* sheet is produced as follows. A sheet is fed from a feeder tray into the mark engine module and moved to the registration component, while a video image is received and laid down onto the continuously revolving photo-receptor belt. As this image and the sheet meet in the transfer component, the image is printed onto the sheet.
Fig. 4. Schematic view of a machine component (inverter).

The sheet is then moved through the inverter (without inversion) to the output on the right and from there into an output tray. For a duplex sheet, two video images are received. After the first image has been printed onto the sheet as for a simplex sheet, the sheet is inverted in the inverter and moved through the duplex loop back to registration. In the meantime, the second video image is laid down onto the belt so that it can be transferred onto the sheet’s back side when the sheet passes through the transfer component. Afterwards, the sheet is inverted again in the inverter (so that it is face-up) and moved out.

2.2. Control software architecture and control process

In modern machines, the hierarchical control software architecture of the reprographic system often mirrors the architecture of the machine. Increasingly, each machine module comes with its own microprocessor, memory, etc., and with software that controls the module’s operations. In more integrated systems, the software at least conceptually is broken down into controllers for different subsystems.

Part of a module controller’s job is to integrate the operations of the machine module into complete functions. For instance, a mark engine module controller (Fig. 5) may “export” exactly two functions, namely printing a simplex or duplex sheet. When told to execute one of these functions at a certain time, the controller will autonomously start and monitor the necessary operations at the right times. Another part of a module controller’s job is to mask local variances in image and sheet processing, in particular timing variances. In other words, under feedback control, a module controller abstracts away many of the local deviations from the expected behavior and thus makes the module’s functions predictable.

On top of these module controllers lies a system controller (Fig. 6) that breaks the system’s functions down into module functions and coordinates the modules in order to produce the desired documents. For example, in order to deliver a set of simplex sheets in a desired output tray, the system controller will tell the feeder, mark engine and finisher modules to feed, print and finish the sheets at certain times such that together a complete document is produced.

This print-engine system controller receives a potentially continual stream of document specifications from a variety of sources, such as the network, the scanner, and the fax input. A document specification only describes the desired output, e.g., “five collated, stapled, double-sided copies of a given high-light color, 10-page document”. This specification is mapped into a sequence of sheet specifications, with specific images on each side, that
must reach the output in a certain order. The system controller’s job is to determine the operations that will produce this sequence, while optimizing machine productivity. From the sheet specifications, the system controller first plans the module operations that need to be executed for each sheet, and then schedules these operations. (By planning, we mean the decision of what operations to execute in what order. By scheduling, we mean the decision of when to execute these operations.) Scheduling operations as close together as possible, even interleaving them when feasible, enables the controller to keep the machine as busy as possible—maximizing productivity for the customer.

In certain common situations, such as when copying or when rendering a long document, the production of the document has to start before the structure or length of the entire document is known to the system controller. This means that planning and scheduling have to start before the entire document is known, and that execution has to start before the entire schedule is known. In other words, planning, scheduling and execution must happen incrementally and concurrently. Thus, the system controller has to be able to both generate and commit to schedules incrementally.
The system controller works under tight real-time constraints, as machines may produce prints at the rate of 60–180 pages per minute. The controller typically gets a few tenths of a second of real time to process each sheet specification.

2.3. Satisfying the machine’s constraints

When planning and scheduling the machine’s operations, the controller has choices in selecting and interleaving operations. It tries to optimize certain criteria, such as the start and completion times of a document, while honoring the modules’ physical and computational constraints.

The transportation and printing of sheets and images is constrained in various ways by the physics of the machine. For example, for a machine to operate properly, sheets are transported in almost continuous movement along the paper path, and the timing of sheets is determined by the lengths and speeds of the transport components; images can be placed on the photo-receptor belt only at certain places (e.g., because a seam in the belt must be avoided); and sheets and images have to be synchronized in the transfer component. The properties of both components and sheets may impose constraints on the execution. For example, an inverter may only be able to invert sheets that don’t exceed a certain length.

Furthermore, it would be simple to transport and print one sheet at a time, but productivity can be improved significantly if multiple sheets are printed in tight succession. In this case, the controller has to make sure that sheets never collide. For example, the time it takes a sheet of paper to be inverted in the inverter is longer than the time for it to just pass through. Thus, if a sheet to be inverted at the inverter is followed by a sheet that is not to be inverted, the controller has to schedule a gap between the two sheets in order to avoid having the second sheet “catch up” with the first one and jam. The length of the gap depends both on the inversion time (which is proportional to the length of the inverted sheet) and on the time it takes to switch the inverter gate.

The print-engine system controller is by far the most challenging piece of software in a reprographic machine. In the past, the construction of such controllers has been a complex and labor-intensive task. Experienced software engineers started from the expected document specifications (e.g., all duplex sheets, or one simplex (cover) sheet followed by all duplex sheets, together with sheet sizes, etc.) and identified a fixed set of operations that would produce sheets according to these specifications on a given configuration. They then analyzed the interactions of machine components during those operations and devised special case rules that would produce optimal schedules for the majority of documents expected (e.g., A4-size sheets, all simplex or all duplex). The outputs of the analysis were detailed flow charts that dictated which parametric template to use for scheduling under which circumstances.

Component interaction analysis is complex. Mapping this directly to control software leads to code that is difficult to understand, maintain, and extend. Most importantly, this practice of manual analysis and code development results in configuration-specific software. Reprographic machines are following the common trend towards plug-and-play systems, where the customers can buy and put together different machine modules to
satisfy different needs. Yet for a machine that is configured by the customer, the system controller has to be itself compositional, something that is almost impossible to provide economically with the traditional approach.

Thus, to facilitate controller development and enable modular machines, we set out to understand the constraints on document production imposed by machine components, to formalize the reasoning that has hitherto been done informally, and to develop algorithms that perform that reasoning in a plug-and-play system.

In the balance of this section, we will develop a domain theory for a model-based version of the print-engine controller.

2.4. Modeling requirements and assumptions

Our domain theory is driven by our primary task, control. By using a declarative modeling approach, however, we are able to extend this theory and thus use our models for other tasks. We will show later what has to be added to this theory in order to enable tasks such as design optimization and productivity analysis.

Conceptually, reprographic machines may be thought of as multi-pass assembly line machines, where parts (e.g., sheets and images) are moved along the assembly line (e.g., paper path, photo-receptor belt), manipulated, and put together, until a desired output is produced. Intuitively, each component is a transducer of timed streams of sheets and images. Thus, it may receive a sheet at a certain time at its input port, transform the sheet as directed by a control command, and produce the result as output at a certain later time at its output port. For example, when directed to invert a sheet, an inverter receives a sheet at its input port, changes its orientation, and forwards the otherwise unchanged sheet to its output port. For a complete machine, producing an output sheet consists of executing a set of component operations that together, if performed in the right sequence, transform input sheets and images into printed output sheets.

Capabilities. Each distinct operation of a component is modeled as a capability. Components can have several capabilities. A component capability is defined by

- the transformation it performs (e.g., "inversion changes a sheet’s orientation from face-up to face-down and vice versa");
- constraints on the features of sheets and images (e.g., "the sheet length has to be less than 436 mm for inversion");
- its timing behavior (e.g., "the time it takes to move a sheet from input to output, with inversion, is the bypass path length plus sheet length divided by component speed");
- and any requirements on internal resources (e.g., "only one sheet can be inverted at a time, and inversion requires that the inversion gate is switched to the inverting position").

To give a first impression of how such a capability is modeled in our modeling language, CDL, the following description defines the inverter with just its invert capability. It can be thought of as a declarative description of a simulation model. This model will be completed and explained in more detail later on.
Component Inverter(int speed) {
  EntryPort in; ExitPort out;
  UnaryResource r_in;
  IntVariable t_out;
  FeatureVariable s_in, s_out;

  Capability Invert(IntVariable
    t_in) {
    in.Input(s_in, t_in); // sheet, times at input
    out.Output(s_out, t_out); // and output ports
    s_in.length <= 436; // feature constraint
    s_out == s_in
    except {orientation}; // sheet transformation
    s_in.orientation
    == 1 - s_out.orientation;
    t_in + (218+s_in.length)/speed
    == t_out; // time constraint
    r_in.Allocate(t_in,
                  s_in.length/speed); // resource constraint
  } // Capability Invert
} // Component Inverter

A component often has multiple capabilities. For example, an inverter can also forward
a sheet unchanged through the bypass path (Fig. 4). Typically, each capability has its own
constraints on features and timing of sheets and images, but uses some shared resources.
A resource may be the space between transport rollers or on a belt, a gate that can be
switched into different positions, or a bin that has a limited capacity. For example, while
the inverter's bypass capability does not change the sheet and requires less time to move
the sheet from input to output, the sheet still needs the rollers while moving through, and
the inversion gate has to be in the bypass position. By modeling rollers and gate as shared
resources—and use of rollers and gate as allocation constraints for these resources—we
are able to model the interactions between capabilities in a modular fashion.

Task. This theory of components and their capabilities is based on a detailed analysis
of the requirements of print-engine control. There are two parts to this analysis,
conceptualization and representation, discussed in the following.

First, our models need to provide the necessary information to enable the control task. As
described, print-engine control consists of two stages, planning and scheduling. Planning
consists of identifying the component capabilities that in sequence produce a desired doc-
ument sheet. The result is a plan of selected component capabilities for one sheet, or simply
a plan. Scheduling means finding feasible timings for the capabilities in such a plan,
resulting in a sheet schedule, a timed sheet plan, that forms the basis for the control com-
mmands (e.g., “feed at time 1500 in tray 1, move at time 3000 in transport 2”, etc.). The total
set of sheet schedules at a specific time is simply called the schedule. While a sheet plan
usually is determined independently for each sheet specification, a schedule may interleave
these plans and therefore must honor additional consistency and resource constraints.
For each sheet, the input to the planning phase are a specification of the desired features of the output sheet, as well as input ports that provide blank sheets and new images (e.g., feed trays and video input). While the machine’s paper and image paths are defined by its components and their connections, planning is somewhat complicated by the fact that there may be multiple alternative paths and loops, and not all components may be able to handle all sheets. In order to find a sheet plan, the planner takes into account each capability’s transformation and feature constraints: the plan must correctly produce the desired output sheet from the input sheet and image(s), and none of the feature constraints (e.g., constraints on sheet size) must be violated. For example, given a well-defined input orientation, the sequence of capabilities along the paper path will determine the output orientation of the sheet by composing the changes of the sheet’s orientation feature along the way. If there are multiple possible plans for a given output specification, either the planner or the scheduler may decide on which one to choose.

Given such a plan (or multiple plans) for each sheet, together with constraints on the sheets’ output order, the scheduler then has to find a time for each selected capability that satisfies all timing constraints and possibly optimizes some objective function (e.g., productivity measure).

Representation. More difficult than to identify what information to capture is to determine how to represent this information. The most important guiding rules have been to start from first principles, and to heed the “no function in structure” principle [5]. While this is common credo in the model-based reasoning and related communities, it was harder to convince software engineers of the benefits. In the following, we go through one real example in particular that illustrates the differences to prior ways of representing constraints and control rules.

Consider first as an example the formulation of the so-called inversion constraint as told to us by engineers: “if a duplex sheet is followed by a simplex sheet, the simplex sheet has to be fed with a delay that is equal to the inversion time of the duplex sheet”. Engineers found it useful to “compile” the system’s expected behavior into a constraint between simplex and duplex sheets, simply because traditionally duplex sheets were always inverted while simplex sheets were not. Thinking about it this way made it easier to write the case-based control software. However, newer machines can deliver documents either face-up or face-down, and thus simplex sheets are sometimes inverted, while duplex sheets sometimes are not inverted after the second image transfer. So it seemed, in order to be more general, that the inversion constraint should be stated as a constraint between inverted and non-inverted sheets. Today’s machines, however, also process multiple sheet sizes (e.g., an 11-by-14 inch (A3) sheet that is folded as an insert in an 8.5-by-11 inch (A4) magazine). Since inversion time is proportional to sheet length, it turns out that the inversion constraint also holds between two inverted sheets if the first one is longer than the second one. This generalization was “discovered” only when we developed our first inverter model from first principles.

Many constraints, in particular timing constraints like the inversion constraint, intuitively are expressed as constraints between interacting components and/or interacting sheets and images. However, this formulation quickly makes it awkward or even impossi-
ble to model machines in a modular way. If the configuration changes only slightly, e.g., if another inverter is added to the paper path (say, before the image transfer), the interaction analysis has to be redone to account for the accumulating delays. What is worse, the inversion constraint may become different for each inverter, depending on the relative position of the inverter in the configuration. Formulating physical constraints as constraints between multiple sheets and images also runs counter to the requirement that the controller be able to schedule sheets incrementally, sheet by sheet, and would further complicate the model as well as the controller implementation. High-end machines, for example, are able to process up to ten different sheet sizes. Analyzing and keeping track of all possible interactions is not an attractive option.

In summary, the original formulation of the inversion constraint was not robust when either the configuration, the sheet behavior, or the sheet properties changed.

Using a modeling approach that derives constraints from the physical structure of devices as shown in Fig. 7 provides a better basis for reusability and compositionality. Starting from first principles, we find, for example, that performing the invert capability has a certain duration depending only on the sheet’s features, and that the capability requires and competes for certain component resources such as roller space and gate position. Similarly, the bypass capability has a certain duration and competes for the same resources. Neither capability has to mention how it interacts with multiple executions of itself or another capability. Instead, we rely on the constraint systems to manage these interactions.

For system control, because of the abstraction provided by the low-level module controllers (cf. Section 2.2), we are able to represent capability execution as discrete events with predictable durations and transport times. We can further model all velocities as either constant or, when required, as changing discontinuously.

2.5. Domain theory

Based on the above discussion, we now present an abstract domain theory for print engines. This theory will be fleshed out and realized in the modeling language presented in the next section.
We model a reprographic machine as a set of connected components. Structurally, a component is described by the tuple \((N, P_i, P_o, R, A)\), where \(N\) is its type, \(P_i\) and \(P_o\) its entry and exit ports, \(R\) the set of internal resources shared by its capabilities, and \(A\) the parameters of the model. For instance, \((\text{inverter}, \{\text{in}\}, \{\text{out}\}, \{f_{\text{in}}, f_{\text{out}}, f_{\text{inv}}\}, \{\text{length, speed}\})\) describes these structural elements of an inverter. While the modeler is free to choose the granularity of components, there typically is an appropriate level that corresponds to the level at which systems are composed by system designers. For example, modeling each roller in an inverter results in details that we can’t make use of in the system controller, while modeling an entire machine module as one component requires the kind of hand-made interaction analysis we want to avoid. (The modeling granularity may also depend on the task, in particular if abnormal component behavior is to be modeled as well.)

Behaviorally, each component is modeled with one or more capabilities, where each capability is a distinct operation of the component, typically on a single sheet or image. A capability is described by the tuple \((U, I, O, C)\), where \(U\) is its control command (naming the capability, with reference time), \(I\) and \(O\) are sets of input and output events, and \(C\) are its feature and timing constraints, \(C_f\) and \(C_t\). An event is a triple \((P; S; T)\), where \(P\) is an entry or exit port, \(S\) is a sheet or image entering or exiting through \(P\), and \(T\) the time of entry or exit. Sheets and images are represented through their features (e.g., length, width, color, orientation, and images). The feature constraints \(C_f\) are constraints on and between the sheet’s or image’s features (combined in \(S\)), while the timing constraints \(C_t\) are constraints on and between the timing variables \(T\). \(C_f\) includes constraints that represent the capability’s transformation. \(C_t\) includes resource allocation constraints.

A composite configuration is defined as a set of components with connections between their ports. When capabilities of two connected components are selected and composed for a sheet plan, the output event of the first component’s selected capability becomes the input event of the second component’s selected capability. Thus, both feature and timing constraints are propagated within a sequence of selected capabilities. In particular, the transformational constraints on sheet and image features are accumulated from input to output, providing a complete specification of the output sheet produced by a sheet plan. (The latter, forward simulation with discrete events and event propagation, is of course available in many formalisms, in particular discrete event simulation languages [2,4,46] and Petri Nets [25,37]. However, these languages are generally restricted to simulation, performance evaluation, and reasoning about specific software properties such as freedom from deadlocks.)

3. Modeling languages

Developing a suitable modeling language for the domain theory was crucial to getting our approach adopted by software engineers. It may be obvious that engineers feel most comfortable with a language that provides domain-specific constructs at the right level of abstraction. Still, in research, we often do not go beyond (or do not publish more than) model representations in a language such as Lisp or Prolog. Instead, the engineer’s modeling language has to have a minimum of ballast needed for reasoning about the
models as well as a familiar look and feel. This means, for example, to write models at
the level of components and capabilities instead of functions with recursive function calls.
Also, moving to a syntax close to C++ lowered the barrier significantly. (One software
engineer called this “a 1000% improvement in readability for people like me” over the
previous Prolog-like syntax.) At the same time, we had certain reasoning capabilities in
mind. In order to be able to simulate and reason about models in this language in a
more general context than control, we based the language on the concurrent constraint
programming (CCP) paradigm.

One might say that the resulting language feels like a domain-specific modeling
language, looks like a procedural language, and behaves like a declarative language.
We call this language the Component Description Language (CDL). This language is
being used regularly by software engineers at Xerox since 1995 and is an integral part
of a generic, reusable machine control toolkit. We would like to emphasize that we
consider both this higher-level modeling language and the lower-level CCP language that
provides its foundation important elements of our approach. They both serve important
purposes, one to support human communication and the other to support computer
processing. In the following, we first present CDL and then show its equivalent in the CCP
framework.

3.1. The component description language

CDL provides behavioral statements (constraints) akin to those available in a typical CCP
language, together with constructs for the specification of structural elements not usually
available in a constraint language. We present the language syntax informally and define
its semantics through the compilation to CCP.

3.1.1. CDL models

Defining a component and its capabilities in CDL is akin to defining an object and its
methods in an object-oriented language. Recall that a component \((N, P_i, P_o, R, A)\) is
defined by its type \(N\), entry and exit ports \(P_i\) and \(P_o\), set of resources \(R\), and parameters \(A\),
as well as its capabilities \((U, I, O, C)\) with control command \(U\) (reference time \(T_r\)), input
and output events \(I\) and \(O\), and constraints \(C = C_f \cup C_t\) (feature constraints \(C_f\) and timing
constraints \(C_t\)). A component definition in CDL follows the following template.

Component N(A, ... ) { 
  declarations for ports, resources, and variables ...;
  Capability U(Tr) { 
    Pi.Input(Si;Ti); ...; Po.Output(So;To); ...; 
    Cf; ...; Ct; ...; 
  } // Capability U
} Component N

As a concrete example, consider again the inverter component (Fig. 4). Structurally, the
inverter has two ports, in and out, through which sheets enter and leave. We model the
rollers at entry and exit ports as (unary-capacity) resources \(r_{\text{in}}\) and \(r_{\text{out}}\), because
only one sheet is allowed in a port at any one time. We also model the inverter switch as a
(state) resource \( r_{\text{inv}} \) that has to be in either “bypass” or “inverting” position while the sheet is moving through. (Constraint systems, including resources, are further explained below. Note that "bypassing" and "inverting" denote values of state variables.) Finally, the model is parameterized by the length of the path from entry to exit ports (in mm), and by the speed of the rollers (in m/s). (Parameters may be instantiated either when defining an instance of the component, when composing components, or when selecting component capabilities at run-time.)

The inverter’s two capabilities may be modeled as follows. A sheet \( s \) to be passed through without inversion (first capability) will enter the component at time \( t_{\text{in}} \) and exit at time \( t_{\text{out}} \). Only sheets of width 285 mm or less can be handled by the inverter. It will take a certain amount of time \( d_{\text{byp}} \) to move from entry to exit, and the sheet will be in the entry and exit rollers for a duration \( d \). These times are determined by the length and speed of the component as well as the length of the sheet. Finally, the two roller resources are busy while the sheet is in the rollers, and the switch resource has to be in bypassing state for the whole time. The component controller associated with the inverter will be instructed to perform this capability with the command \( \text{Bypass}(t_{\text{in}}) \) (name and reference time).

A sheet \( s_{\text{in}} \) to be inverted (second capability) will be transformed to an output sheet \( s_{\text{out}} \) that is identical to the input sheet except for its orientation, which is reversed. In addition to the width constraint, sheets are limited to a length of 436 mm. Also, the time between entry and exit increases by the time it takes to invert the sheet, which is proportional to its length. Resource allocations correspond to those of the bypass capability. The corresponding control command is \( \text{Invert}(t_{\text{in}}) \).

The complete CDL model of the inverter is defined as follows.

```java
Component Inverter(int length, int speed) {
  EntryPort in; // ports
  ExitPort out;
  UnaryResource r_in, r_out; // declarations
  StateResource r_inv;
  IntVariable t_out, d,
      d_byp, d_inv;
  FeatureVariable s, s_in, s_out;

  Capability Bypass(IntVariable t_in) {
    in.Input(s, t_in); // input/output events
    out.Output(s, t_out);
    s.width <= 285;  // feature constraint
    t_in + d_byp == t_out; // event time constraints
    d_byp == length/speed;
    d == s.length/speed;
    r_in.Allocate(t_in, d); // resource constraints
    r_out.Allocate(t_out, d);
    r_inv.Allocate(t_in, d_byp,
        "bypassing");
  } // Capability Bypass
```
A module is modeled by specifying its components and their connections, and by defining itineraries, the mappings from module commands to component commands. A module integrates the control of its components into higher-level commands: when the control module receives a command, it sends the required component commands to its components. Composite modules currently do not have their own resources (because we did not find a need for it). They may pass parameters through to the components.

For example, we may define a "long inverter", a simple module that consists of a transport component connected to an inverter. The transport component has an interface similar to the inverter, but only one capability, moving sheets, similar to the inverter's bypass capability. Let the speed be a parameter of the module, while the length of the components (in mm) becomes fixed in the definition. This would be defined in CDL as follows.
As a more complex example, consider the mark engine configuration shown in Fig. 5, which consists of a transfer component, an inverter, merge and split gates, and two transport components (for registration and duplex loop).

Component MarkEngine(int speed) {
  EntryPort in, video; // ports
  ExitPort out;
  MergeGate merge(); // components with
  SplitGate split(); // lengths
  Transport registration(218, speed),
    loop(654, speed);
  Transfer transfer(); Belt belt(speed);
  Inverter inverter(218, speed);

  in == merge.lower; video == belt.video; // connections
  out == split.lower;
  merge.out == registration.in;
  registration.out == transfer.in;
  belt.image == transfer.image;
  transfer.out == inverter.in;
  inverter.out == split.in;
  split.upper == loop.in;
  loop.out == merge.upper;

  Itinerary Simplex(IntVariable t) {
    merge.MergeLower(_); registration.Move(_);
    belt.Compose(_);
    transfer.Mark(t); inverter.Bypass(_);
    split.SplitLower(_)
  } // Itinerary Simplex

  Itinerary Duplex(IntVariable t1, IntVariable t2) {
    merge.MergeLower(_); registration.Move(_);
    belt.Compose(_);
    transfer.Mark(t1); inverter.Invert(_);
    split.SplitUpper(_);
    loop.Move(_);
    merge.MergeUpper(_); registration.Move(_);
    belt.Compose(_);
    transfer.Mark(t2); inverter.Invert(_);
    split.SplitLower(_)
  } // Itinerary Duplex
} // Component MarkEngine

Finally, for convenience, one may also define higher-level constraints in CDL and then use those constraints in component models. For example, the constraints that relate entry time, capability duration, exit time, and resource allocations often depend on length and speed of the path. One could replace them by a single “shift” constraint that would be defined as follows (cf. the inverter model above).
Constraint Shift(IntVariable t1, IntVariable t2, IntVariable d, FeatureVariable s, UnaryResource r1, UnaryResource r2) {
    t1 + d == t2;
    da == s.length/speed;
    r1.Allocate(t1, da);
    r2.Allocate(t2, da);
} // Constraint Shift

Using the shift constraint, the inverter’s bypass capability would be simplified to the following.

Capability Bypass(IntVariable t_in) {
    in.Input(s, t_in); out.Output(s, t_out);
    s.width <= 285;
    Shift(t_in, t_out, length/speed, s, r_in, r_out);
    r_inv.Allocate(t_in, length/speed, "bypassing");
} // Capability Bypass

In summary, these device models in CDL are direct descriptions of the capabilities of system components, i.e., they directly capture, at an appropriate level of abstraction, the physics of the underlying components. Models are declarative specifications of the channels of communication between the components and their environment (structure), as well as the constraints to be satisfied whenever components interact with their environment (behavior).

3.1.2. Constraint systems

We provide three constraint systems in CDL to support the domain theory described above. First, we use an integer constraint system to represent times, lengths, etc. We provide the usual equality and disequality constraints on arithmetic expressions with the expected interpretation (e.g., \(x + 2*y \geq z\)). Note that the standard units used for times and lengths are milliseconds and millimeters, and since no smaller resolution is required and only discrete behaviors are modeled so far, an integer representation is sufficient for our purposes.

We use a feature constraint system to describe and reason about the attributes of sheets and images. Feature variables allow for a hierarchical, extensible representation of these attributes. For example, the feature variable \(s\) may have features length, width, orientation, front image, back image, etc., denoted by \(s.length, s.width, \) etc. Features denote other variables or constants. Some features, such as \(s.frontImage\), may themselves be feature variables, with subfeatures such as \(s.frontImage.position, \) etc. Others, such as \(s.length\), may refer to integer or other variables. New features may be added dynamically. The only feature constraints are \(s1 == s2\) and \(s1 == s2\) except \(e\), where \(s1\) and \(s2\) are feature variables, and \(e\) is a list of feature names. The intended interpretation of the equality with exception is that \(s1\) and \(s2\) are identical except for the features listed in \(e\). In other words, any feature added to \(s1\) or \(s2\) is propagated to the other variable unless mentioned in \(e\). This allows one to express transformations in capabilities: the statements
say that sheet \( \text{so} \) is identical to sheet \( \text{si} \) except that the orientation is reversed (where 1 stands for face-up and 0 for face-down). In addition, features may be used in other constraints like normal variables of their respective types. For example, the integer constraint \( \text{s.width} = \leq 285 \) states that the width of sheet \( \text{s} \), an integer variable, has to be less than 285 mm.

CDL also includes a resource constraint system to describe how a capability uses certain resources. Different types of resources are supported:

- unary-capacity resources: allocations cannot overlap in time;
- multi-capacity resources: allocations can overlap so long as the total amount at any time does not exceed a capacity limit;
- state resources: allocations can overlap only if they require the same state.

Thus, multiple uses (or allocations) of a resource are restricted according to the resource type. For example, assume that a capability moves a sheet through a roller at time \( t \) for duration \( d \), and that sheets cannot overlap in the roller. The roller can be represented by a unary-capacity resource \( r \), and the requirement can be expressed by the constraint \( r.\text{Allocate}(t,d) \). As another example, assume that a capability requires a switch in a certain position \( s \) at time \( t \) for duration \( d \). The switch can be represented by a state resource \( r \), and the requirement can be expressed by the constraint \( r.\text{Allocate}(t,d,s) \).

A multi-capacity resource could be an output tray that can hold a limited number of sheets at any time. In planning and scheduling, different components make allocations for different resources. The constraint system constrains the timing variables (here \( t \) and \( d \)) in accordance with the resource constraints of the resource type.

### 3.2. Modeling with concurrent constraint programs

Rather than develop an interpreter or reasoning tools directly for CDL, we have chosen to translate CDL into CCP, using CCP as a generic framework for this language [21]. From CCP, CDL inherits a well-defined, logical, and extensible semantics. This semantics and the interpreter build on experience with constraint and logic programming languages. The simple but powerful CCP framework also supports declarative reasoning techniques such as composition, deduction, abduction, and partial evaluation, techniques that are well-known and documented in the logic programming and constraint programming literature (e.g., [5,29,33,39,45]). By translating CDL models to concurrent constraint (CC) programs, we can put these techniques to use on our domain-specific models, e.g., to compose components to modules and machines, to simulate models, to reduce model size and pre-compile model execution, and to derive additional information from models.

We would like to stress this point further. CDL is parametric with respect to constraint systems, i.e., new constraint systems can be integrated seamlessly into the language as needed, a property it inherits from the CCP framework (see below). Thus, beyond the particular constraint systems provided in CDL, we will point out the kind of reasoning enabled by the semantics and generic operations over constraint systems that are available in CCP.
3.2.1. The CCP framework

CCP is a general framework for computation based on the computational interpretation of a fragment of logic [41]. Arising from concurrent versions of logic programming, it can be seen as a general, clean version of forward chaining languages that have been used as a basis for problem solving in engineering systems.

Like constraint logic programming, CCP is built on a two-level logical framework. At the base level is the notion of constraints. A primitive constraint over a given set of variables is a partial specification of the values the variables can take. A typical example of a constraint is a first-order formula over some algebraic structure, e.g., \( x - 3 \geq y, y \geq 0 \).

A crucial relation between constraints is that of entailment: a set of constraints \( c_1, \ldots, c_n \) entails a constraint \( c \), if whenever each of the \( c_i \) holds \( c \) holds as well. For instance, \( x - 3 \geq y, y \geq 0 \Rightarrow x \geq 3 \) for all values of \( x \) and \( y \). Thus, constraints can combine additively, without any prejudice about their source, to produce other constraints. A set of primitive constraints together with an entailment relation is called a constraint system. CCP is parametric with respect to constraint systems: as long as a constraint system provides an entailment operation, it can be integrated seamlessly in the declarative and computational framework of CCP.

Constraints are viewed as the primitive assertions around which programs in CCP are built. CCP provides the following basic programming constructs: the ability to assert a constraint to the constraint store (tell), the ability to check if the set of constraints in the store entails a given constraint (ask), the ability to run multiple such programs in parallel (parallel composition), and the ability to introduce a new local variable on which constraints can be asserted (hiding or scoping).

Programs are defined by clauses \( H::S \) with head \( H \) and body \( S \). Fig. 8 defines the abstract syntax of a CC program.

Note that variables \( x_1, \ldots, x_n \) in the head of a clause \( p(x_1, \ldots, x_n)::S \) are implicitly universally quantified. We also require that no more than one such clause may be defined for a name \( p \).

\[
\begin{align*}
\text{Program } P & := H::S | P \cdot P \quad \text{(Head and body)} \\
\text{Statements } S & := c \quad \text{(Tell)} \\
& | \ [A] | [A; S] \quad \text{(Conditional)} \\
& | S, S \quad \text{(Conjunction)} \\
& | x^*S \quad \text{(Local variable)} \\
& | H \quad \text{(Process call)} \\
\text{Ask } A & := C \Rightarrow S | A \quad \text{(Ask-tell)} \\
\text{Condition } C & := c | c, c \quad \text{(Ask)} \\
\text{Process head } H & := p(x_1, \ldots, x_n) \quad \text{(Name and arguments)}
\end{align*}
\]

Fig. 8. Abstract CC program syntax.
Our concrete syntax is similar to the abstract syntax. Naming follows Prolog conventions (lower-case for constants, upper-case for variables, the character “_” for the anonymous (nameless) variable). The conditional \[ [C_1 \rightarrow S_1; C_2 \rightarrow S_2; \ldots; S] \] is written as

```latex
\begin{align*}
  &\text{if } C_1 \text{ then } S_1 \\
  &\text{elseif } C_2 \text{ then } S_2 \\
  &\text{else } S
\end{align*}
```

The syntax of constraints \( c \) depends on the constraint system. The main differences to CDL are that “=” instead of “==” is used for equality, and the resource variable in allocations is moved to the first argument of the allocation (e.g., \( r.\text{Allocate}(t,d) \) becomes \( \text{allocate}(r,t,d) \)). Variable hiding is done implicitly. We further assume the usual data structures known from logic programming, including lists \( L \) whose head \( H \) and tail \( T \) can be identified by the operation \( L = [H|T] \).

This CCP language, in contrast to many modeling languages, including CDL, is a generic programming language. It does not provide any domain-specific or physics-related notions, neither structural ones (e.g., the notion of “components”, “model fragments”, or “connections”) nor behavioral ones (e.g., the notion of physical behaviors represented by differential equations or even the idea of time).

Computation of CC programs progresses by monotonically accumulating constraints in the store, and by checking whether the store entails constraints. (Generally, it is the programmer’s responsibility to avoid over-constraining the store and thus causing failure.) Synchronization between processes is specified by the ask construct: a conditional \( [C_1 \rightarrow S_1; C_2 \rightarrow S_2; \ldots; S] \) suspends until one of the ask constraints \( C_i \) is entailed by the constraint store, or all ask constraints become disentailed (i.e., their negations become entailed). In the former case, \( S_i \) is executed, in the latter case \( S \). For example, the process

```latex
p(x,y) :: \text{if } x \geq 3 \text{ then } z = 0.
```

when called as \( p(x,y) \), will reduce to the conditional and suspend. When the constraints \( x-3\geq y, \ y\geq 0 \) are told to the store by another process, the ask constraint becomes entailed by the store, and the constraint \( z=0 \) is told. If instead a constraint \( x=2 \) is told, the conditional reduces to the empty process. Fig. 9 formally defines the semantics of a CC program. The program state is denoted by the tuple \( S, s \), which consists of the current statements (or goals) \( S \) and the constraint store \( s \). Program execution starts with a (goal) statement and an empty store, and is represented as a sequence of program states.

CC computation is done in a parallel context: multiple processes concurrently tell and ask constraints on shared variables. This naturally supports compositionality: new processes can be added to a computation without further changes, as all communication happens indirectly through constraint variables. There needs to be no pre-specified division of responsibility between different processes about who is a “producer” and who is a “consumer”.

Further, CC programs are declarative, i.e., they can be read as logical assertions: the ask operation corresponds to implication, the hiding operation to existential quantification, and parallel composition to conjunction. As mentioned before (also \([11,12,22,26]\)), this has enabled the development and adaptation of various reasoning techniques, such as partial evaluation, program transformation, and abstract interpretation.
3.2.2. Modeling with CC programs

Intuitively, for our application, the behavior of a device can be represented in CCP as alternative assertions, each corresponding to a capability. Each alternative waits for its control command and expected input events; it then asserts the capability’s feature and timing constraints and forwards the output events. For each port, the process defines an argument variable (that is shared with a connected process when used in a configuration of components). Another argument variable is used to receive control commands. These argument variables serve as message channels across which the component processes pass commands and events.

Thus, in order to model a capability \( \langle U, I, O, C \rangle \) of component \( \langle N, Pi, Po, R, A \rangle \) \( (C = Ct \cup Ct) \), a process can be defined using the following template.

\[
\begin{align*}
N(U, I, S, O, R, A, ...) &::=
\begin{cases}
\text{if } (U = [U | UsT], I = [(Si, Ti) | IsT], ...) \text{ then } & (C, \% \text{ constraints on } Si \text{ and } So) \\
& (Ct, \% \text{ constraints on } Ti, To, \text{ and } R) \\
& (O = [(So, To) | OsT], ...), \\
& N(UsT, IsT, ..., OsT, ..., R, ..., A, ...) \\
\text{else if } ... \end{cases}
\end{align*}
\]

Us is the control channel that receives commands \( U \) for the capabilities. Is and Os are the input and output channels through which events \( (S, T) \) are received and forwarded. Us, Is and Os are streams, and the result of computation is a set of extensional streams that, together with the asserted constraints, defines the state of the model over time. After executing a capability, the process is called recursively. Multiple alternative capabilities are concatenated in one conditional with alternatives.

Note that such a model \( M \), given control command \( U \) and inputs \( I \), computes outputs \( O \) and accumulates constraints \( C \) by executing the subprocess corresponding to the capability selected by \( U \). Logically,

\[
M \vdash U, I \rightarrow C, O.
\]
Thus, given the expected commands and inputs, the model computes exactly the constraints on the store specified by the capabilities \((U, I, O, C)\) of the modeled component.

The skeleton shown above is the target template for compiling CDL models to CC programs and thus defines the semantics for executing behaviors described by such models. We can go further in our combination of CDL and CCP, though. Many reasoning techniques, such as partial evaluation, abduction, or declarative debugging, are difficult to control for general-purpose CC programs. The issue is that there are normally multiple possible behaviors, and no guidance for which one to produce, nor stop rules for exiting loops. In a typical implementation, a user is asked to help in the process. By making use of the domain-specific knowledge available in the CDL models, and by knowing the task, guidance rules and an objective function can be developed. For example, the objective for model pre-compilation may be to unfold all capabilities but not other processes, or all feature constraints but not the timing constraints. This kind of structural knowledge is available in or can be inferred from CDL models. Our approach is therefore for the CDL-to-CCP compiler to automatically add suitable annotations and parameters to the CC programs, and to modify reasoners such that they can take advantage of these annotations (see below). Reasoning tools that provide hooks for user interaction (e.g., partial evaluators [50] and declarative debuggers [17,44]) can be readily adapted in this way by using the annotations instead of asking the user for guidance. Note that annotations, in particular structural annotations (e.g., which CC program corresponds to a component or module), can also be used to translate modified CC programs back to CDL models.

3.2.3. CC models

We can now show the expansion of the complete model of the inverter component in our CCP language (cf. Fig. 4) from the CDL model of the inverter presented above. The modeled behaviors are identical; the differences are in headers, conditionals, and recursive calls (shown in slightly larger font size).

```plaintext
inverter(Us, In, Out, R_in, R_out, R_inv, Length, Speed) ::
  if (Us = [bypass(T_in)|UsT], In = [(S,T_in)|InT]) then {
      % on command and input event for bypass capability
      S.width <= 285, % feature constraint
      T_in + D_byp = T_out, % time constraints
      D_byp = Length/Speed,
      D = S.length/Speed,
      allocate(R_in, T_in, D), % resource constraints
      allocate(R_out, T_out, D),
      allocate(R_inv, T_in, D_byp, bypassing),
      Out = [(S,T_out)|OutT], % output event
      inverter(UsT, InT, OutT, R_in, R_out, R_inv, Length, Speed)
  }
  elseif (Us = [invert(T_in)|UsT], In = [(S_in,T_in)|InT])
    then {
      % on command and input event for invert capability
```
S_in.width <= 285, % feature constraints
S_in.length <= 436,
S_out = S_in except [orientation], % sheet transformation
S_in.orientation = 1 - S_out.orientation,
T_in + D_inv = T_out, % time constraints
D_inv = Length/Speed + D,
D = S_in.length/Speed,
allocate(R_in, T_in, D), % resource constraints
allocate(R_out, T_out, D),
allocate(R_inv, T_in, D_inv, inverting),
Out = [ (S_out, T_out) | OutT ], % output event
inverter(UsT, InT, OutT, R_in, R_out, R_inv, Length, Speed).

A module is defined as a set of connected components. As mentioned, components are connected through parallel composition of their CC models and shared event-channel variables. The unconnected component ports become the ports of the module. In addition, recall that a module integrates the control of its components into higher-level commands.

Modules can easily be modeled as CC processes as well. Recall as example the module with transport and inverter components. The following CC program models the composite module. This program is equivalent to the CDL model LongInverter presented above.

long_inverter(Us, In, Out, Speed) ::
transport(TUs, In, T2I, _TR_in, 436, Speed),
inverter(IUs, T2I, Out, _IR_in, _IR_out, _IR_inv, 218, Speed),
long_inverter_send(Us, TUs, IUs).

long_inverter_send(Us, TUs, IUs) ::
if Us = [simplex(T)|UsT] then {
    TUs = [move(T)|TUsT], IUs = [bypass(_)|IUsT],
    long_inverter_send(UsT, TUsT, IUsT)
}
elseif Us = [duplex(T)|UsT] then {
    TUs = [move(T)|TUsT], IUs = [invert(_)|IUsT],
    long_inverter_send(UsT, TUsT, IUsT)
}.

When called, the program long_inverter creates the two connected component processes as well as a recursive process that receives module commands and sends out component commands according to the specified itineraries.

Finally, higher-level constraints such as the shift constraint above (Section 3.1) are compiled trivially to CC processes:
shift(T1, T2, D, S, R1, R2) ::
T1 + D = T2,
Da = S.length/Speed,
allocate(R1, T1, Da),
allocate(R2, T2, Da).
3.2.4. Simulation

The above inverter model can be used directly for simulation, e.g., to compute the modeled effects. For example, in order to get a collection of the inversion constraints, we may send an `invert` command and a variable input event (sheet and time) to the inverter:

```plaintext
:: inverter(Us, In, Out, R_in, R_inv, Length, Speed),
   Us = [invert(T_ref)|_], In = [{S_in,T_in}|_].
```

The resulting constraint store, pretty-printed here for convenience, contains the following variables and constraints.

```plaintext
Us = [invert(T_in)|_],
In = [{S_in,T_in}|_],
Out = [{S_out,T_out}|_],
S_in = {width = W, length = L, orientation = O_in
   |S_out except [orientation]},
S_out = {width = W, length = L, orientation = O_out
   |S_in except [orientation]},
W <= 285,
L <= 436,
O_in = 1-O_out,
T_in + D_inv = T_out,
D_inv = (Length+L)/Speed,
T_ref = T_in,
R_in = {allocate(T_in,D)|_},
R_out = {allocate(T_out,D)|_},
R_inv = {allocate(T_in,D_inv,inverting)|_},
D = L/Speed
```

We may further set the length and speed variables by adding constraints in parallel to the model in order to get a picture of the timing constraints:

```plaintext
:: inverter(Us, In, Out, R_in, R_out, R_inv, Length, Speed),
   S_in.length = 218, Length = 218, Speed = 1,
   Us = [invert(T_ref)|_], In = [{S_in,T_in}|_].
```

This refinement is sufficient to determine and visualize the precise relative resource allocations, which is useful for informally validating the models. Fig. 10 shows the resource allocations for the inverter’s bypass and invert capabilities.

![Fig. 10. Inverter resources with allocations for the bypass and invert capabilities selected at different times (each rectangle is an allocation).](image)
Fig. 11. Inverter resources with allocations for multiple selected capabilities (two bypasses, two inversions, and two bypasses; note that allocations in $R_{\text{in}}$ and $R_{\text{out}}$ don't overlap, and that different state allocations in $R_{\text{inv}}$ don't overlap).

Fig. 11 shows the allocations for a document executing two bypass operations (allocations 1 and 2), two inversions (3 and 4), and two bypass operations (5 and 6). (Observe the effect of the inversion constraint on the gap between sheets 4 and 5.)

4. Model-based configuration

4.1. Model life cycle

Models used for reprographic machine control play a role in three phases of machine construction and use:
- development—component models are build and debugged;
- configuration—module models are transformed, embedded, and composed;
- control—machine models are used to plan and schedule the system’s capabilities.

Each phase has different requirements on models and reasoners, determined by both human factors and automation needs. In the development phase, engineers choose component models from libraries, build new models, and assemble these to hierarchical module models in CDL. In the course of software development, engineers may also execute, debug and analyze the models as needed. This activity happens in parallel to the design and development of the real (physical) objects. Models are tested in different configurations, and are used to evaluate productivity, sensitivity, complexity, and other control issues.

Engineers develop machine modules, which customers buy and put together to complete machines with the desired functionalities. Thus, the configuration phase is a process extended over two subphases: the module models are embedded in the machine module control software and then used later to configure the system control software when the composite machine configuration becomes known.

Finally, as described before, the control phase involves planning and scheduling the machine’s capabilities, using the models’ constraint-based representation to control, correctly and efficiently, the machine’s operations.

In the balance of this section, we will discuss reasoning techniques for configuration, specifically partial evaluation of CC programs through deduction and abduction. We will address reasoning techniques for control in the next section. (The life cycle outlined above is being applied in a current Xerox product program. While model-based configuration and control as described in this and the next section is fully deployed, not all the techniques
for model development presented below have been transferred to the product program yet. In particular, instead of composing module models from component models, engineers currently develop module models directly.)

4.2. Deriving capabilities through partial evaluation

In the following, we focus on a reasoning procedure that is the core of several model processing steps in our model life cycle, namely the derivation of the composite capabilities of a given model. The question is: given a model describing a module or machine composed of components, what are the composite capabilities of that module or machine given the capabilities of its components? This step is applied when simplifying hierarchical module models to flat module models before embedding them in the control software, when pre-computing all sheet plans after machine composition, and when generating one sheet plan for a specified output. We present and discuss a basic reasoning technique for this step in detail in order to show how CCP can be embedded in a task-specific environment.

4.2.1. Requirements

Recall that a CC model $M$, given control commands $U$ and inputs $I$, computes outputs $O$ and accumulates constraints $C$ by executing the processes corresponding to the component capabilities selected by $U$. Given such a model $M$, the task is to derive the set of capabilities $C$ defined by

$$C = \{(U, I, O, C) | M \vdash U, I \rightarrow C, O\}.$$  

Conceptually, capability derivation is accomplished by performing fold/unfold transformations on the composite model. In principle, a single clause representing a composite configuration is replaced by a clause with a collection of alternative statements, one for each consistent, complete, and minimal selection of component capabilities. A selection is consistent if its constraints are consistent. A selection is complete if, for each selected component capability, the inputs are produced by the connected upstream components (if any) and the outputs are consumed by the connected downstream components (if any). A selection is minimal if it cannot be decomposed into other, complete capabilities. (For instance, the sequence of two simplex capabilities is not minimal, while a duplex capability is minimal, even though some component capabilities, such as inversion, are contained twice for execution at different times. A simplex capability without, say, a selected transfer capability would be incomplete.)

Given our modeling approach, the reasoning technique for capability derivation is essentially partial evaluation of CC programs. The derivation process, however, is complicated by a number of problem and modeling properties. First, components are modeled as endless processes, and the configuration may contain loops (e.g., the duplex loop), but we are only interested in complete and minimal capabilities. Second, during partial evaluation, we typically do not know the necessary inputs, and sometimes we want to reason from a given output. In other words, in addition to deducing outputs from inputs, we sometimes have to abduce inputs from outputs. Finally, what statements are useful to partially evaluate is task-specific; for example, for planning, we may want to evaluate feature constraints, but not timing constraints.
4.2.2. Deduction and abduction

To deal with these issues, we have built a partial evaluator for capability derivation from CC models that uses deduction for forward evaluation of eligible statements, abduction for “backward evaluation” of eligible conditionals, backtracking to explore alternative choices in abduction, and special initialization and termination procedures.

Deduction is defined by the semantics for CC programs (cf. Fig. 9). The constraint store resulting from the evaluation contains the transformed model. Also, for the application to planning, timing constraints are not interpreted.

We define a single abduction step for CC programs, namely abduction over conditionals, as follows. Given a suspended conditional \([A; C \rightarrow S]\) in a quiescent computation (i.e., its ask constraints are not entailed by the constraint store and computation does not progress), if the ask constraint \(C\) is consistent with the store, assume \(C\) by adding it to the store. This abduction step requires a single addition to our semantics of CC programs:

\[
\text{Ask abduction} \quad \frac{(\langle S, [A; C \rightarrow S]\rangle, s) \not\rightarrow \neg s \cup C \land \text{false}}{(\langle S, [A; C \rightarrow S]\rangle, s) \rightarrow (\langle S, s \cup C\rangle, s)}
\]

(1)

Again, the constraint store resulting from the evaluation contains the transformed model.

However, not all conditionals should be eligible for abduction. In particular, we do not want to indiscriminately abduce component capabilities, as this can lead to incomplete and non-minimal capabilities. Intuitively, like deduction, abduction should follow the flow of events, albeit in reverse direction (“generating an input from an output”). Notice that for deducing component capabilities, we have the built-in requirement that a control command and all necessary inputs have to be present. Similarly, for abducting a component capability, we use the heuristic that at least the control command or one input or output of this capability have to be present. Together, these requirements for deduction and abduction will unfold the component capabilities of a composite model as long as there is at least one initial command, input or output present at the model’s interface. Furthermore, the abduction heuristic will lead the partial evaluator to trace the flow of events through the model efficiently and in a way that guarantees complete and minimal composite capabilities. They will be complete because if a capability’s inputs or outputs are present, the capability will be evaluated. They will be minimal because only capabilities that are “required” because of an existing input or output are evaluated.

A remaining issue of the abduction heuristic is that “control command” and “capability input and output” are domain concepts. While they can be identified easily in the CDL model, they are indistinguishable from other statements in a generic CC program. For this purpose, the compiler annotates conditionals with the heuristic. Such annotations also allow us to control the abduction of other “types” of conditionals (e.g., to control recursion). Thus, such annotations enable domain-specific reasoning within a domain-independent modeling language.

We augment our CCP syntax with the annotated ask \(C \rightarrow N S\):

\[
\text{Ask} \quad A \ ::= \ C \rightarrow S \mid C \rightarrow N S \mid A; A \quad \text{(Ask-tell)}
\]

\[
\text{Annotation} \quad N \ ::= \ c \mid c; c \quad \text{(Constraints)}
\]
The concrete syntax for annotated conditionals \([C_1 \rightarrow N_1 \; S_1; \; C_2 \rightarrow N_2 \; S_2; \; \ldots; \; S]\) is

\[
\begin{align*}
\text{if} & \; \; C_1 \; \text{abduceif} \; N_1 \; \text{then} \; S_1 \\
\text{elseif} & \; \; C_2 \; \text{abduceif} \; N_2 \; \text{then} \; S_2 \\
\text{else} & \; \; S
\end{align*}
\]

Concretely, a capability \((U, I, O, C)\) of component \((N, P_1, P_0, R, A)\) is compiled to an annotated conditional as sketched in the following pseudo-code.

\[
\begin{align*}
N(U, I, S, \ldots, O, \ldots, R, \ldots, A, \ldots) & \; : \; :
\text{if} & \; \; (U = [U|USt], \; I = [(Si,Ti)|ISt], \; \ldots) \\
\text{abduceif} & \; \; (U = [U|USt]; \; I = [(Si,Ti)|ISt]; \; \ldots; \; O = [(So,To)|OSt]; \; \ldots) \\
\text{then} & \; \; (C_f, \; \% \; \text{constraints on Si and So} \\
\text{Ct}, \; \% \; \text{constraints on Ti, To, and R} \\
O & \; \; (O = [(So,To)|OSt], \; \ldots, \; N(USt, \; ISt, \; \ldots, \; OSt, \; \ldots, \; R, \; \ldots, \; A, \; \ldots) \\
\text{elseif} & \; \; \\
\end{align*}
\]

The semantics of ask abduction is replaced by the following rule:

\[
\begin{align*}
((S, [A; \; C \rightarrow N \; S]), s) & \; \rightarrow _- \\
\text{Ask abduction} & \; \; N = \emptyset \; \vee \; \exists \; c \in N \; \rightarrow \; s \vdash c \; \; s \cup C \not\vdash \text{false} \; \quad (3)
\end{align*}
\]

This computation rule, together with the compiler’s use of annotations for commands, inputs and outputs, implements the capability abduction heuristic. In the following, we discuss some application and implementation details for capability derivation with the procedures just described.

4.2.3. Initialization, design restrictions, termination

We use capability derivation both to plan for a given output sheet specification and to find the plans for all possible sheet specifications. In the former case, the model’s output is instantiated with an event consisting of a concrete sheet specification and a variable time. In the latter case, where no explicit output is given, we initialize derivation by placing a generic event (with variable sheet and time) as “seed” at the model’s output port. This is akin to standard logic programming evaluation of declarative programs, where arguments may be either given to check for consistency, or left open to generate instances. In both cases, our derivation procedure has a first output event that enables abduction in the last component and is propagated through the network of components as described above.

Since a module or machine model is often composed from generic component models found in a library, the composite model may define composite behaviors that should be avoided for a particular configuration. This may include certain component capabilities that should not be used, certain sheet sizes that may not be available, and unnecessary or potentially infinite loops of selected capabilities. Constraint programs make it possible to
simply add design restrictions in the form of additional constraints on program variables. With CC programs, some of these restrictions may be defined as programs running in parallel with the model, “monitoring” its streams (e.g., command channels), and telling constraints that restrict partial evaluation when appropriate. For example, the following program can be run in parallel with a component process.

```plaintext
restrict_itinerary(Us, Command, N) ::
    if Us = [U|UsT] then (
        if U = Command then (
            N > 0,
            N1 is N - 1,
            restrict_itinerary(UsT, Command, N1) )
        else
            restrict_itinerary(UsT, Command, N)
    ).
```

When running this as, say, `restrict_itinerary(Us, invert(_), 1)` sharing the inverter’s control channel `Us`, the partial evaluator will deduce the invert capability at most once, i.e., the derived capabilities will contain at most one inversion. Generic programs such as `restrict_itinerary` can themselves be predefined in a library, and engineers can specify the use of design restrictions together with either the models or the configuration software.

Finally, when a complete composite capability has been derived, the resulting program state is `(S,s)`, where `S` are the unevaluated statements and `s` is the constraint store. `S` contains statements that were ineligible for evaluation, as well as suspended conditionals (e.g., the recursive continuations of component processes). `s` contains all evaluated constraints, including inputs, outputs, and feature constraints. A final, task-specific termination step will clean up and extract the necessary information and record it as composite capability `(U, I, O, C)` for the intended purpose. Again, this step can make use of compiler annotations in order to distinguish between the different types of statements and constraints in the program state.

4.2.4. **CDL compilation revisited**

It is important to remember that the derivation procedure developed above is by itself a task-independent reasoning technique. It becomes a task-specific procedure through the addition of domain-specific knowledge (e.g., what is a capability) in the form of generic annotations (e.g., what can be abduced) by the CDL-to-CCP compiler. Before presenting the complete capability derivation procedure, we review the compiler’s role.

When compiling a component, the compiler adds the following information to the generated CC program.

- **Statement type**: annotations classifying statements as eligible or ineligible for partial evaluation as explained above.
- **Abduction**: annotations on conditionals for abduction as explained above.
- **Context information**: various kinds of information (such as component ID, depth, etc.) for debugging and logging.
- **Helper code**: CC programs generated from additional specifications such as the design restrictions mentioned above.
4.2.5. Derivation procedure

Recall that a component \( (N, P, P_o, R) \) is modeled and compiled to program \( M \) with interface \( N(Us, Is, \ldots, Os, \ldots, R, A) \). We first define a procedure \( \text{derive} \) which returns a composite capability \( (U, I, O, C) \) for output specification \( S \) at port \( P_o \) of model \( M \):

\[
(U, I, O, C) = \text{derive}(M, P_o, S);
\]

get channel \( Os \) of port \( P_o \) of model \( M \);

partially evaluate \( (Os = [(S,\_)]_\text{\_}\_ \_\_, N(Us, Is, \ldots, Os, \ldots, R, A)) \);

extract capability \( (U, I, O, C) \) from the result;

return \( (U, I, O, C) \).

The procedure for deriving all capabilities for a composite model is defined as follows.

\[
\mathcal{C} = \text{derive-all}(M);
\]

for each port \( P_o \) of model \( M \):

for each \( (U, I, O, C) = \text{derive}(M, P_o, \_\_) \):

record capability \( (U, I, O, C) \) in \( \mathcal{C} \);

return \( \mathcal{C} \).

Consider as an example the model for \( \text{long_inverter} \), consisting of a transport and an inverter. Deriving its capabilities by partially evaluating the statement

\[
(\text{Out}=\{(\_,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, \text{long_inverter}(Us, In, Out, Speed))\}
\]

(where the first statement is the “seed” that starts the abduction) results in the following capabilities \( (U, I, O, C) \), pretty-printed here for convenience.

**Capability 1:**

\[
\{\text{long_inverter}:\text{simplex}(TT_in),
\text{long_inverter}.trp:move(TT_in),
\text{long_inverter}.inv:bypass(IT_in)\},
\{\text{long_inverter}.in:<(S,TT_in)\}
\{\text{TT_in} + 436/\text{Speed} = \text{TT_out}, \%
\text{from transport}
\text{TT_out} = \text{IT_in}, \%
\text{from connection}
\text{S.width} <= 285, \%
\text{rest from inverter}
\text{IT_in} + \text{D_byp} = \text{IT_out},
\text{D_byp} = 218/\text{Speed},
\text{D} = \text{S.length}/\text{Speed},
\text{allocate}(R_{in}, IT_{in}, D),
\text{allocate}(R_{out}, IT_{out}, D),
\text{allocate}(R_{inv}, IT_{in}, D_byp, bypassing))\}
\]

**Capability 2:**

\[
\{\text{long_inverter}:\text{duplex}(TT_in),
\text{long_inverter}.trp:move(TT_in),
\text{long_inverter}.inv:invert(IT_in)\},
\{\text{long_inverter}.in:<(TS_in,TT_in)\}
\{\text{long_inverter}.out:<(IS_out,IT_out)\},
\]
\[
\begin{align*}
&\text{TT_in} + 436/\text{Speed} = \text{TT_out}, \quad \text{from transport} \\
&\text{TT_out} = \text{IT_in}, \quad \text{from connection} \\
&\text{TS_in.width} \leq 285, \quad \text{rest from inverter} \\
&\text{TS_in.length} \leq 436, \\
&\text{IS_out} = \text{TS_in} \text{ except [orientation],} \\
&\text{TS_in.orientation} = 1 - \text{IS_out.orientation}, \\
&\text{IT_in} + D_{\text{inv}} = \text{IT_out}, \\
&D_{\text{inv}} = 218/\text{Speed} + D, \\
&D = S_{\text{in}.length}/\text{Speed}, \\
&\text{allocate}(\text{R}_{\text{in}}, \text{T}_{\text{in}}, D), \\
&\text{allocate}(\text{R}_{\text{out}}, \text{T}_{\text{out}}, D), \\
&\text{allocate}(\text{R}_{\text{inv}}, \text{T}_{\text{in}}, D_{\text{inv}}, \text{inverting}) >
\end{align*}
\]

Note that these capabilities are still parameterized by machine parameters (such as the speed of the module) and by run-time parameters (such as the length of the sheet). In our implementations so far, the capability constraints \( C \) are always reduced to primitive constraints before they are embedded into machine modules. These constraints can be translated to a constraint network to be solved by a generic constraint satisfaction algorithm, and under certain assumptions can even be pre-compiled to fixed control procedures at configuration time (see below). Thus, in considerable part due to the separation of models from control code, and because of the flexibility to transform models to different formats, we impose few restrictions on the controller implementation, and this approach integrates well with a procedural machine control environment.

In a later extension, we may allow \( C \) to contain complex statements, including remaining conditional statements that have to be evaluated at configuration time or even run time. This requires a full CC program interpreter in the control software, but enables considerable flexibility in customizing its behavior.

4.2.6. Complexity analysis

As in most real-life planning problems, the complete derivation process is exponential in the number of machine modules, as the number of possible machine capabilities increases exponentially as machine modules are added. For instance, composing three machine modules with five capabilities each results in up to 125 machine capabilities. Large machines may have thousands of machine capabilities. Not all compositions usually are possible, though. For instance, module capabilities that process different sheet sizes can’t be composed. In practice, we found that the exponentiality of the derivation process is not as big an issue as the memory required to store all of the resulting machine capabilities. The latter issue can be addressed in different ways, e.g., by composing only a subset or certain parts of the capabilities at configuration time and the rest on demand. Still, as indicated before, the planner eventually has to be able to find machine capabilities within a few dozen milliseconds, and pre-computing all commonly used capabilities is currently the only way to guarantee that these real-time requirements are always satisfied. The real-time requirements also distinguish this problem from most traditional planning and scheduling problems, where run-times of seconds, minutes or even hours usually are considered acceptable.
5. Model-based control

When composing machine modules to a complete machine, the modules pass up their module models to the system controller, where the models are composed to a machine model as explained above. (Some customization, such as instantiation of the speed parameter, may be done at this time.) At run time, given this machine model and the specifications for desired document sheets, the system controller plans the module operations that need to be executed for each sheet, and then schedules these operations (cf. Fig. 12).

In this section, we give an overview of the controller’s tasks and discuss two alternative scheduling methods in more detail. Unless indicated otherwise, all algorithms and processes have been implemented for product programs that have been launched or are under development.

5.1. Planning

The planning problem is defined formally as follows: given an output specification $S$ (defined by feature constraints) for a machine output port $P_o$, find one or all plans of machine capabilities that produce output $S$. Given machine model $M$, the procedure $\text{derive}(M, P_o, S, (U, I, O, C))$ will return such a plan, a derived capability $(U, I, O, C)$. Alternatively, if all machine capabilities have been derived from $M$ at configuration time as described above, the planner can select one of these by searching for one with an output at port $P_o$ that matches (or unifies with) specification $S$. The controller may also check whether all necessary inputs $I$ are available and ready in the machine, searching for alternative capabilities if necessary.
For example, assume the two capabilities for long_inverter discussed above, with Speed set to 1 m/s, and a sheet specification \( S \) with length 218 mm and width 282 mm. Then the first capability, if selected, would be instantiated and reduced to the following.

```
Capability 1:
<long_inverter:simplex(TT_in), long_inverter.trp:move(TT_in), long_inverter.inv:bypass(IT_in),
<long_inverter.in:(S, TT_in)),
<long_inverter.out:(S, IT_out)),
TT_in + 436 = TT_out,
IT_in + 218 = IT_out,
allocate(R_in, IT_in, 218),
allocate(R_out, IT_out, 218),
allocate(R_inv, IT_in, 218, bypassing)>
```

Thus, partial evaluation of the machine model is akin to compiling the model to parameterized plans. As parameters (such as the sheet length) are set through capability selection, capability selection is akin to plan instantiation.

When a machine capability, or sheet plan, has been selected, the planner will forward the capability, in particular its control commands and timing constraints, to the scheduler. As the planner receives a stream of output specifications, it forwards a stream of sheet plans to the scheduler.

5.1.1. Complexity analysis

If sheet plans (machine capabilities) have been derived at configuration time (off-line), planning reduces to selecting a plan that matches the sheet specification. This is linear in the number of sheet plans if implemented naively, but efficiency can be improved significantly if plans are classified and stored in a hierarchical fashion according to sheet features (e.g., discriminating by sheet size first).

5.2. Scheduling

The scheduler receives the stream of sheet plans, to which it typically adds further constraints, such as precedence constraints between output times (to enforce the correct sheet output order) and between the current (real) time and a plan’s input times (to force the plans to be scheduled in the future) [20]. The scheduler optimally solves these timing constraints in order to find a feasible schedule. A typical objective function for the optimization is \( t_n \) or \( t_n + t_1 \), where \( t_i \) is the output time variable of sheet plan \( i \), and \( n \) is the number of sheets being scheduled.

The scheduler may solve the timing constraints repeatedly in order to generate a schedule incrementally. A sheet plan’s constraints may be solved immediately after receiving it, or when several plans are available. Solving the constraints instantiates the reference time variables of the control commands, which are then sent to the module controllers in order to execute the schedule.

There is a spectrum of scheduling algorithms and architectures that can be based on this framework. We have implemented algorithms that range from efficient search engines.
using pre-compiled versions of the constraints (running in launched products) [42], to flexible, reactive optimizers based on a generic constraint solver (deployed in a product under development) [20]. These algorithms further allow for various choices such as the amount of look-ahead in the sequence of selected capabilities, and the time of when to commit to parts of the incremental schedule [19]. In the following, we describe two scheduler implementations. (Detailed algorithms and test results are beyond the scope of this paper. Instead, we refer the interested reader to the various cited publications for more information.)

5.2.1. Compiling constraint solving to a finite-state machine

In practice, for some configurations, it is possible to pre-compile part of the scheduler’s work at development or configuration time. The pre-compilation enables better response time at the expense of flexibility. Several schemes have been worked out and implemented (e.g., [18,42]). We sketch one of them [18].

Suppose we have a schedule for \( i = 1 \) sheets and are interested in all possible ways of scheduling sheet \( i \) that may eventually lead to an optimal schedule for \( n \) sheets \( (n \geq i) \). Because the order of sheets is constrained, the output time \( t_i \) of sheet \( i \) has to be greater than the scheduled output time \( t_{i-1} \) of sheet \( i-1 \). Note that scheduling sheet \( i \) immediately after sheet \( i-1 \) may not necessarily lead to an optimal schedule; leaving a gap may be better if a future sheet can make use of that gap and would otherwise have to be scheduled much further out. (For example, when printing a simplex sheet followed by a duplex sheet, we can delay the simplex sheet and feed the duplex sheet first. After the front page of the duplex sheet has been printed and while this sheet is going around the duplex loop, we can feed and print the simplex sheet. The simplex sheet exits the machine first as desired, and the total print time is the time of printing a duplex sheet only.)

Assume that none of the machine’s capabilities contains inequality timing constraints, i.e., each capability has a finite extent in time, a finite execution time (e.g., the time from feeding to outputting the sheet). Let the largest such extent of any machine capability be \( r_{\text{max}} \) milliseconds. Since scheduling sheet \( i \) more than \( r_{\text{max}} \) apart from the current schedule would only leave unproductive holes in the schedule and thus never lead to an optimal schedule, \( r_{\text{max}} \) effectively places an upper bound on the output time of sheet \( i \). In other words, \( t_{i-1} < t_i \leq t_{i-1} + r_{\text{max}} \).

Assume further that sheets can only be scheduled at discrete intervals. For example, a typical configuration places a constraint on the timing of images, as images can only be placed in a finite and relatively small number of slots on the photo-receptor belt. (If such a constraint doesn’t exist, it can be added artificially to restrict the problem to a finite number of states.) As a consequence, given a schedule for \( i-1 \) sheets, there is only a finite number of schedules for sheet \( i \) that can lead to optimal schedules.

Not all of these schedules for sheet \( i \) are consistent with the existing schedule. The schedule for sheet \( i \) may interleave with the existing schedule, and the resource constraints determine which interleavings are allowed. Note, however, that the schedule for sheet \( i \) overlaps with the existing schedule only to a certain extent. In fact, at most the last \( r_{\text{max}} \) milliseconds of the existing schedule are affected, since the schedule for sheet \( i \) can reach

---

Footnote: Papers are available either from the respective proceedings or from the first author.
at most that much back into the existing schedule. Thus, we need to take into account only the ending of a schedule, at most the last $r_{\text{max}}$ milliseconds, in order to determine how it can be extended optimally to one more sheet.

These insights are the basis for an algorithm that analyzes the timing constraints of all capabilities to determine $r_{\text{max}}$ and other parameters, and then generates all possible schedule endings as well as all possible extensions from these endings for all possible capabilities. Each extension leads to a new schedule with its own ending. Scheduling endings and extensions become the states and transitions of a configuration-specific, non-deterministic finite-state machine (FSM). For all possible schedules and sheet specifications, the FSM contains at least one and possibly multiple transitions that may lead to optimal schedules. Remember that in general it isn’t possible to determine an optimal schedule until all sheets are known. Given an (initially empty) schedule, the scheduler can explore alternative schedules incrementally, in parallel, and in a bounded space, committing only to potentially optimal schedules. When the entire document is known, or when requested to make a decision, the scheduler can then simply pick the best of these schedules.

One limitation of this method is that it only applies to a certain class of configurations, namely those that can be modeled without inequality constraints and have only a small number of useful schedules per capability. Another disadvantage is that it customizes the scheduler to a particular configuration that can only be changed in restricted ways later on, unless the FSM is regenerated (which currently takes on the order of hours for typical configurations and has to be done off-line). Another issue is the size of the FSM; while a typical departmental multi-function device results in only a few hundred transitions, the FSM quickly explodes for larger configurations. The main advantage of the FSM-based method, however, is that it is guaranteed to generate optimal schedules for the currently known sheets in bounded time. It has been shown to be effective for the intended class of configurations and is used in commercial products.

5.2.2. Adapting a generic constraint solver

To overcome the limitations of the FSM-based scheduling approach, in particular to make the generic scheduler software available to a larger range of products and pose no restrictions on plug-and-play, our second-generation scheduler uses a generic, real-time constraint solver to optimally schedule sheet plans.

Using a generic constraint solver reduces the scheduling problem to a familiar approach: build a constraint network from the timing constraints of the selected capabilities and search for a solution. The main issue becomes how to adapt traditional constraint-solver technology to the sort of embedded, ongoing computation required for reactive scheduling [20].

Our scheduling task is representative of a class of reactive controllers that have a model of the controlled system and thus can predict the system’s behavior. All other information, such as information about documents, is disclosed incrementally. We found three points particularly relevant for the constraint solver:

- The solver constantly alternates between adding constraints, searching for solutions, and committing to partial solutions (e.g., the next sheet to be printed) [19] (which makes memory management, in particular garbage collection in the constraint network, more difficult).
The solver has to distinguish between temporary decisions (for search) and committed decisions (parts of a schedule that are being executed).

The solver has to manage the relation between timing variables and real time (e.g., the timings for any given sheet have to be greater than or equal to the current real time—which is a moving target—until the sheet is being printed).

We used two guiding principles in our solver design. First, all low-level constraint operations (propagation, search, garbage collection) should be as incremental and distributed over time as possible in order to minimize their effects at any one time and to allow for trade-offs between memory and processor usage. Second, the scheduling algorithm should be able to make use of its application knowledge, which, together with a well-defined model of reactive computing, helps the solver manage its resources effectively. We also provide special functions for our reactive scheduling task, such as the ability to constrain variables with respect to the current real time when required and easily remove this constraint when appropriate.

An on-line scheduler, one that makes choices in parallel to job submission and schedule execution, is inherently suboptimal, as it makes decisions based on incomplete information about the future. Besides the implementation of the underlying constraint solver, there are various task-level choices that can affect the efficiency and optimality of a generic on-line scheduler. We discuss two: the timing of commitments, and adding redundant constraints.

A traditional constraint-optimizing scenario is to assert the constraints and then search for a solution for all variables, optimizing an objective function. The scenario of on-line scheduling requires repeatedly finding a solution for only a small subset of the variables, such as those in the sheet plan to be executed next. While this solution should be part of an optimal solution for all variables (all known sheets), we want to keep the other variables open in case more information about future sheets becomes available. In other words, the optimizer is to return a solution for the next sheet only, but guarantee that it is part of a currently optimal solution for all known sheets. We found that this approach, full optimization with minimal commitment, leads to the best overall productivity in an on-line scenario (on average about 5% worse than the theoretical optimum for randomly generated jobs). This idea can be encapsulated in a new enumeration primitive that generates an optimal solution for all variables, but instantiates only those variables that have to be returned for execution of the next sheet plan [19]. A real-time constraint solver as described so far has been implemented and deployed for a product under development.

Another way to approach the optimality of off-line scheduling with an on-line scheduler is to add redundant constraints. In particular, we have investigated how to exploit domain knowledge to improve optimization performance of a branch-and-bound search algorithm, in terms of both efficiency and productivity [24]. We developed a taxonomy of redundant constraints that distinguishes global, local and incremental constraints along one axis, and job-specific and job-independent constraints along the other. We found that adding domain-specific redundant constraints can have a significant impact on optimization performance. Incremental constraints in particular project tight lower and upper bounds on the objective function based on an analysis of previous solutions to a subset of the variables, which can be effective in supporting branch-and-bound search. We also found that there is a trade-off between finding a first solution soon and finding the optimal solution when using branch-and-bound search. Redundant constraints generally lead to a better first solution,
but it may take longer to find that first solution due to the added overhead. The biggest challenge of applying redundant constraints is the automatic generation and verification of such constraints, still largely an unsolved research subject.

As these different examples of scheduler implementations indicate, our approach allows software engineers to make decisions about the desired degree of flexibility to react to dynamic changes, and to scale the resulting software to different speed and memory requirements. A model-based approach to control does not preclude the tuning of the control software for a particular context.

5.2.3. Complexity analysis

When analyzing the complexity of a constraint solver-based scheduling algorithm, we consider two scenarios: the complexity of typical configurations, and the complexity of CDL models in general. In general, finding a solution for a constraint network is exponential in the number of variables. However, the constraints for those configurations we have encountered so far result in constraint graphs that have a tree structure. It has been shown that if the constraint graph has a tree structure and the network is arc-consistent (the domains of variables connected through constraints are made mutually consistent) [36], then a solution can be found without backtracking [16]. Finding an optimal schedule for general sheet sequences (e.g., with mixed simplex and duplex sheets in the same document) is still exponential. For homogeneous sheet sequences and typical machine configurations, however, the first solution is guaranteed to be optimal. In other words, for a set of typical machines, we can always find a solution in polynomial time, and for a set of typical jobs, we can also find the best solution in polynomial time.

In addition to this problem-specific analysis, we can also ask the more general question: what is the problem complexity of any model we might want to represent in CDL? Recent research has shown that there are certain properties of constraint relations which are sufficient to ensure tractability, regardless of the associated problem structure [30]. Currently, all constraints used in our models are either binary equality constraints \( x = y + c \) (for variables \( x \) and \( y \) and constant \( c \)), binary disequality constraints \( x > y \), or unary constraints, or they can be mapped to these constraints (e.g., allocation). (Conditionals are evaluated before scheduling.) These constraints all fall into the class of “max-closed” constraints, which is an algebraic closure condition that ensures tractability [31]. Thus, our current models fall into the class of tractable problems, which confirms the problem-specific analysis.

5.2.4. Prototyping schedulers in CCP

As a final point, while our production schedulers are implemented in a procedural language, we found CCP an effective means for prototyping scheduling algorithms and strategies, and we made use of this a number of times [19, 24]. For example, the reactive scheduler can be modeled in CCP as a process that accepts messages \( \text{sheet\_plan}(S, I) \) and \( \text{schedule\_request}(T, A) \). \( \text{sheet\_plan}(S, I) \) indicates a new sheet plan \( S \) with index \( I \). \( \text{schedule\_request}(T, A) \) demands a scheduled event \( A \) for current real time.

---

5 The subsequent arguments have been worked out by Lisa Purvis (Xerox WCR&T, internal communication).
T; A is either the index of a sheet plan to be executed, or empty. The main scheduler process can be defined as follows.

\[
\text{scheduler}([M|Ms], V) ::= \\
\hspace{1em} \text{if } M = \text{sheet\_plan}(S,I) \text{ then } \{ \\
\hspace{2em} \text{constrain\_sheet}(S, I, V, W), \\
\hspace{2em} \text{scheduler}(Ms, W) \} \\
\hspace{1em} \text{elseif } M = \text{schedule\_request}(T,A) \{ \\
\hspace{2em} \text{minimize\_and\_select}(V, T, A, W), \\
\hspace{2em} \text{scheduler}(Ms, W) \}.
\]

\text{constrain\_sheet}/4 asserts the timing constraints for the given sheet plan I and an existing set V of known variables, resulting in the new set of variables W. \text{minimize\_and\_select}(V, T, A, W) returns, for the list V of variables and a current time T, a scheduled event A and remaining variables W. (See [19] for more details.)

This dual use of the CCP framework, for modeling both hardware and software, proved very useful in rapidly prototyping the various elements of our approach. The CCP framework provided a convenient environment for prototyping the domain theory, for developing the constraint systems, and for experimenting with using the models for control. At the same time, clearly defined interfaces between the models and the control software, as presented above, ensured that the same models and algorithms could be used in both the prototypes and the real, embedded implementation.

6. Related work

Model-based computing has its roots in model-based reasoning. Most generally, work on model-based reasoning has investigated relationships of structure, behavior and function of physical devices, with the intent of deriving expert knowledge about device behavior from a “first principles” description of the device [1,9,51]. Compositional modeling has long been at the heart of this work [13]: constraints and differential equations have typically been used to describe component or process behavior. Work in this field focuses on two broad areas—the development of notations and tools for modeling physical systems, and the development of reasoning algorithms (e.g., diagnosis algorithms) for using these models. Along the first dimension, various languages have been developed for the (usually qualitative) description of physical systems, e.g., the language of Qualitative Process Theory (QPT) [14], Qualitative Simulation (QSIM) [34], and the Device Modeling Environment (DME) [35]. Recent consolidation of this line of work has yielded the design of the language CML [6].

Our work shares many of these ideas. One difference is that we base our modeling language on the simple and mathematically well-developed framework of concurrent constraint programming. With recent extensions for discrete and continuous time modeling [27,43], concurrent constraint programming offers a powerful modeling framework for modern control and diagnosis applications. Particularly attractive here is the dual reading of declarative CC models as executable programs, which enables the use of the large body of implementation techniques developed in the context of concurrent logic programming.
However, CCP does not come with any domain-specific constructs. This has to be added for a particular domain (as done through CDL), or programmed as part of the models. Whether this is seen as an advantage or disadvantage depends on the context of use. In our context, we found that CDL was easily accepted and customized for different tasks expressly because it often only required a stripped-down constraint solver and very customized reasoners to do a task. In fact, CDL has been out of our hands for several years and has been maintained and adapted to new needs by product engineers mostly without our involvement.

We have tried to point out at various places that CDL can benefit from the program manipulation techniques available for CCP. One place where this is visible is that CDL does not require the modeler to state low-level inference steps such as how to reason from output to input that have been necessary in other modeling languages (e.g., [10]) that have been compared to CDL.

Also, the model-based computing framework emphasizes the integration of information generated from models (using fairly general-purpose reasoning techniques) with conventional software architectures and systems (e.g., a real-time procedural scheduler). Thus, we address all elements of the model life cycle, from modeling language to reasoning to task execution. A similar completeness can be found in the work on model-based programming for autonomous systems at NASA [52]. Here, we put more emphasis on generic compositional modeling and reasoning principles, while NASA's work has emphasized model-based reactive control.

Further related work of interest is the automatic, model-based generation of low-level control code, such as the component and module controllers that receive the control commands generated by the system controller. For the most part, these controllers are still being developed by hand in today's commercial products. Automatic generation of low-level code for copier image and paper paths has been investigated both from hybrid CC models [7,28] and from qualitative models (using QPT as the underlying theory) [40]. The latter research has resulted in the Knowledge Intensive Engineering Framework (KIEF), an integrated framework for deriving behavior from function models and for generating control sequences from behaviors for individual actuators (motors, etc.) by adding timing information. The resulting control sequences are not reactive and cannot be composed or interleaved easily. (In fact, the approach can neither represent nor reason about constraints between multiple sequences.) Still, controllers generated in KIEF are orthogonal to our framework and probably could be integrated into it as the module controllers used by the system controller.

Our work also overlaps with modeling approaches used in Object-Oriented Analysis and Design (OOA/D), and with work on Domain-Specific Software Architectures (DSSAs). OOA/D often starts from a domain analysis, resulting in an informal or semi-formal classification that reflects the physical world, and from which software objects and methods are derived [8]. Objects contain both data and task-specific functions, and in recent years it has been recognized that such software components can't be developed independently, but that the patterns of interactions among objects have to be designed as well [23].

The idea of DSSAs is generally to first construct a reference architecture or framework, which software developers specialize by using domain models and task-specific requirements to guide the selection or building of additional software components [3]. Work in
this area has focused strongly on Architecture Description Languages that allow one to
describe the behaviors or functions of software components.

Both OOA/D and DSSA are complementary to model-based computing. They (mostly)
do not address the formal, task-independent modeling of physical devices, and where they
use models, these are tied strongly to the code (i.e., they do not separate machine-specific
and task-specific components). However, these methodologies have proved useful when
developing generic, task-specific architectures like the ones needed for our application.

Constraint-based scheduling has long been used in “intelligent” scheduling systems, as
an alternative to classical Operations Research algorithms (e.g., for job-shop scheduling),
and often in combination with AI-style search, rule-based and constraint-satisfaction
algorithms [38,53]. The most general approach to constraint-based scheduling is using
a constraint (logic) programming language, as proposed for cc(FD) [47], CHIP [48] and
related languages. These languages typically provide several basic built-in constraint
systems with generic, powerful constraint solving algorithms, and allow one to define
new, domain-specific constraints on top of these. Conceptually, we follow a similar
approach, building on a well-defined semantics with flexibility in language extension and
program manipulation. However, little research has been done in high-level modeling with
constraint programming languages. For us, the constraint programming language is the
“assembly language”, upon which we have built a modeling language with its own set of
analysis and translation tools.

7. Model multi-use: Extending the reach of model-based computing

An important argument for model-based reasoning has long been “multi-use”, or the
re-use of models for multiple tasks. Indeed, while plug-and-play, automatic configuration,
and optimal control are the immediate benefits of our general approach, model multi-use is
an important long-term benefit in our domain. CCP supports multi-use in several ways: by
enabling declarative and thus task-independent models, through its parametric framework
that allows one to easily define new constraints and even new constraint systems, and
through its tool box of standard symbolic program manipulation techniques. This gives
CDL the genericity and extensibility needed for model multi-use.

In this section, we report from preliminary experience with multi-use for three different
tasks: module simulation, productivity analysis, and design optimization. For each task,
we also point out how models, reasoners, and task framework had to be changed from
the control task. (Detailed algorithms and test results are beyond the scope of this
paper. Instead, we refer the interested reader to the various cited publications for more
information.6)

7.1. Module simulation

In practice, the system controller is often developed before the machine modules and
their controllers are available. In order to test the system controller in the context of
the complete machine control software, it is usually desirable to simulate at least the high-level interface behavior of the module controllers. This behavior typically is defined by the communication protocol between system and module controllers (e.g., a hand-shaking protocol), and by the module’s available capabilities (including commands and timing). While the former is given by the architecture and fixed for all modules, the latter information can be derived from the CDL models.

Thus, given a module model, it is possible to automatically generate interface code that simulates the module controller’s communication behavior as experienced by the system controller. This module controller simulator accepts the system controller’s commands, simulates the execution of these commands, and interacts with the system controller in other ways (e.g., returning module status information on demand). This task requires no changes to the models and the task framework set up for the control task, only new reasoners to generate the simulators. This has been implemented in a product program for use by software engineers.

7.2. Productivity analysis

Early in product development, system engineers often use engineering judgment and experience with past machines to decide on an initial machine design. This design is then evaluated against a set of criteria such as productivity and cost. For high-end reprographic machines, productivity is one of the most important criteria, as it is the strongest determiner of the machine’s value as perceived by the customer. Productivity can be defined as the actual number of pages printed in a given time. (This is in contrast to the rated speed. For example, a 60 pages-per-minute machine can’t actually produce a single duplex sheet in 2 seconds, since moving through the paper path, in particular the duplex loop, adds time.) Productivity analysis evaluates the productivity of a given design on a set of representative documents. The set of representative documents, also called the job demographics, is defined by the anticipated use of the machine as determined by market studies. The job demographics specifies the number of pages in a typical document, the expected sheet sizes, the ratio of color vs. black-and-white pages, etc. One way to represent the job demographics is as a set of document specifications with associated probabilities [32].

Productivity analysis essentially requires a scheduler in order to determine the timings of documents produced on a given machine. In the past, no scheduling software was available at that early stage, and so engineers built simplified, special-purpose schedulers or used spreadsheets to model a design. Also, a design was often analyzed with respect to a simplified subset of the job demographics only, e.g., only homogeneous documents or even just single sheets. With CDL, system engineers are able to quickly model a new design and then use the generic scheduler to analyze this design with respect to a large set of documents. This task requires no changes to the models and the reasoners (for composing models, etc.), while the task framework is changed to one for off-line scheduling and schedule analysis for a set of documents [32]. This has been prototyped and demonstrated for a number of products under development, but is still in an exploratory stage.
7.3. Design optimization

Designing modern electro-mechanical devices is a non-trivial task, exacerbated by ever increasing functionality and performance requirements. Often, engineers have to decompose these concerns when developing a design, for example evaluating cost and productivity independently instead of making a trade-off analysis. And while a rough design, e.g., functionality and abstract component structure, is often easy to come by, there still exist many possible designs in the space of parameters like location, size and speed of the components.

To address this issue, we have investigated how to automatically generate designs from the requirements [32]. We are particularly interested in design optimization from parameterized designs: given the structure of a machine with a set of design variables with unknown values, a solution to the design problem is an assignment of values to these variables such that the functionality requirements (e.g., types of documents to be printed) are satisfied and the objectives (e.g., productivity and cost) are optimized.

The main extensions to CDL required for design optimization were the declaration of properties and design constraints [32,49]. Properties are variables and functions, such as cost, that may be used to evaluate a design with respect to the requirements. For example, a transport component model may define the cost of the component in terms of the length of the component, with the length being a design variable. Design constraints primarily define allowable designs by constraining the design variables. Design variables may be model parameters, such as speed and length, as well as properties. They are the variables to be instantiated by the design optimizer.

We have used these models experimentally in two ways [32]. One is a standard search approach that enumerates consistent solutions for the design variables and evaluates them with respect to the objective function. The objective function includes cost and productivity on the specified job demographics. However, one concern about generating designs that are optimal with respect to the given job demographics is that the job demographics often is a best guess at a characterization of the market requirements. Generally, we believe that it is more useful to characterize the design space relative to a requirement space instead of finding an optimal design for a particular, somewhat arbitrary set of requirements. What can be expected from market studies is a qualitative “feel” for job distribution, resulting in broad classifications such as “frequent” and “infrequent”. Therefore, in a second approach to design optimization, we allow the engineer to specify the job demographics in such qualitative terms. With these, the design optimizer performs a sensitivity analysis to identify a set of distinct, quantitative ranges for job distribution together with designs that are optimal in each range. This has been prototyped and demonstrated on some sample configurations, but is still in an exploratory stage.

8. Conclusions

As the demand for providing more functionality for less cost increases, developing correct and effective software for electro-mechanical systems becomes both more important and more difficult. In this paper, we have presented an instance of model-
based reasoning, model-based computing, a methodology for constructing such software
that is based on a separation of task and machine-specific information. This methodology
starts from declarative, compositional models of a machine’s components, uses generic
software architectures and algorithm, and provides formal reasoning techniques linking
models and architectures. A central element of our approach is the use of concurrent
constraint programming languages and techniques, which constitute powerful tools during
software development and still integrate well with a traditional procedural implementation
environment. Model-based computing enables the automatic composition of software for
multiple systems and multiple tasks.

Consistent with long-standing motivations and claims of model-based and qualitative
reasoning research [10,13,15,51], we see four primary benefits from the use of model-
based computing:

- **Collaborative and concurrent engineering.** Models are formal representations of
  relevant operational information. A development process based on models can result
  in improved quality of hardware and software through better communication between
  engineers, more accurate analysis, and the use of established knowledge repositories.
  Equally important, through the simulation and analysis of new components, model-
  based computing allows for the concurrent development of hardware and software,
  and for the modular development of complex systems.

- **Plug-and-play.** As models are compositional and separated from algorithms, software
  is no longer configuration-specific. Compositional software enables higher product
  flexibility and supports a larger number of possible configurations.

- **Time-to-market.** Once a generic architecture is in place, only models have to be
  changed to obtain control software as the product goes through several prototyping
  cycles or new modules are created. This results in faster system development through
  higher automation, easier adaptation, increased re-use, and improved robustness of
  hardware and software.

- **Multi-use.** Where it is possible to specify declarative models independent of an
  application, models allow for higher synergy between engineering tasks, such as
  design, analysis, control, and diagnosis. (While we and others have demonstrated that
  this is possible in principle, the integration of these tasks through model multi-use has
  been realized to only a very limited extent.)

So far, we have evidence for the first two points and to a limited extent for the third
point from real, commercial product programs at Xerox. For example, regarding improved
quality and communication, engineers in a past product program found errors in their prior
analyses after starting to model components from first principles (cf. Section 2.4). Also,
in various personal reports to us, system-control engineers in a recent product program
using CDL mentioned improved communication with module-control engineers as well
as with system engineers (analysts). Regarding concurrent engineering and development
speed, the system-control engineers in a recent product program were able to test their
software by simulating the modules from their models before the hardware was available.
Also, as hardware specifications kept changing over the course of the program, informal
observation suggested that it was much easier to keep up with the changing hardware than
in previous programs. Finally, regarding plug-and-play, the number of different machine
configurations and capabilities available to the customer from a current product program
by mixing different feeder, mark engine and finisher modules is expected to be at least an order of magnitude larger than ever before.

We have applied model-based computing techniques to the development of control software for reprographic machines. In our experience, the crucial element in such an application is the domain theory, which must offer a way of thinking about machines that enables engineers to formally describe them, and at the same time must enable the automatic use of the resulting models. We would like to emphasize that this approach does not have to result in a monolithic technology that requires changes to an existing software development process wholesale. Instead, we have had best results when integrating various elements incrementally. For example, early in the project, we focused on supporting the software engineers and their immediate needs. There, our approach to technology transfer can be described as “up-stream integration”, i.e., we first concentrated on algorithms and the representation of pre-analyzed models in the procedural target language. Only later did we add the higher-level modeling language and transformation tools that enable the automatic configuration of the control software.

We are currently investigating how our approach extends to design generation, productivity analysis, and on-line and off-line diagnosis. As more and more applications like these benefit from explicit machine models, modeling will become a core activity in the development of computationally controlled systems.

Acknowledgements

The work reported here has been carried out in collaboration with a number of other researchers and engineers within Xerox Corporation. We wish to thank the other past and present members of the Machine Control project (Vineet Gupta, Tim Lindholm and Pierre Berlandier) at PARC; Paul Rulli, Marc Webster, John Conley, Lisa Purvis, and Gregg Bonikowski at the Wilson Research Center in Rochester, NY; and Sue Layer and Jim Matysek within a Xerox Business Division in Rochester. Bob Bauer, Johan de Kleer, and Bob Brookes deserve special thanks for supporting the project in its formative stages. We also thank the reviewers of this paper for their detailed comments.

References

[23] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns—Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1994.