Model-based Configuration of Machine Control Software

Extended Abstract

Markus P.J. Fromherz
Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA
fromherz@parc.xerox.com

Following the trend of mass customization, reprographic machines (photocopiers, printers, fax machines, etc.) are increasingly designed and assembled from standard components. For example, a system engineer may design a paper path for a new print engine configuration from components such as sheet transports, sheet inverters, gates, registration units, and trays. Similarly, customers may choose from and plug together different complex modules such as mark engines and feeder and finisher modules in order to get the desired functionality. One of the most complex steps in this composition of reprographic modules is the automatic configuration of the system control software. In this extended abstract, we give an overview of our approach to this task. See (Fromherz, Saraswat, & Bobrow 1999) for a complete review.

Model-based computing

When planning and scheduling the machine's operations (e.g., feeding, moving, printing, inverting sheets), the system control software has to satisfy each module's physical and computational constraints (timing, sheet sizes, etc.), and take into account the often complex interactions of the modules' behaviors. For this task, we have deployed an approach to developing real-time system-level controllers for electro-mechanical machines that consists of three concurrent activities: (1) the development of application-independent, declarative, constraint-based models of physical machine modules and configurations, (2) the development or re-use of a separate, configuration-independent control software architecture, and (3) the development or re-use of mediating reasoners that provide the glue for embedding the models into the control architecture. In this model-based computing approach, 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.

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. In fact, developing a suitable modeling language for this domain was crucial to getting our approach adopted by software engineers. We learned that engineers feel comfortable with a language that provides the domain-specific constructs important for modeling, at the right level of abstraction, with a minimum of ballast needed for reasoning about the models, and with a familiar look and feel. The resulting Component Description Language, CDL, is a high-level, engineering-oriented language with domain-specific constraint systems. CDL models describe a device's capabilities through input/output transformations and constraints on moving parts, timing, and resources.

However, it can be difficult to provide a simple semantic interpretation to a modeling language with domain-specific constructs, precluding the use of powerful tools to reason about the constraint representations. Our approach is to translate CDL into the lower-level, but very general framework of concurrent constraint programming (CCP) (Saraswat 1993) (cf. the appendix). 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.

CCP makes it natural to build simulation models of physical systems. Intuitively, a device capability can be represented as a set of constraints to be asserted if the expected control command and input events are received. A device with multiple capabilities is defined as a recursive process with multiple, alternative assertions. Connected devices are modeled as processes that communicate events through shared variables. Together, this is the target structure for the CDL-to-CCP compiler. As a simple example, consider the model of a sheet transport component with a single capability, namely moving a sheet from input to output. It receives control commands Us and input events In (sheet S and time T.in), asserts constraints on sheet width, input/output timing and space resource R, and generates output events Out (again sheet and time). The model is parameterized by the component's length and speed.
transport(Us, In, Out, R, Length, Speed) ::
if (Us = [move(T.in)][UsT],
    In = [(S,T.in)][InT]) then (
    S.width <= 285,
    T.in + Length/Speed = T.out,
    allocate(R, T.in, S.length/Speed),
    Out = [(S,T.out)][OutT],
    transport(UsT, InT, OutT, R, Length, Speed)
).

Model-based software configuration

In the following, we focus on the auto-configuration of the system control software after a customer has bought and put together a machine. Each machine module comes equipped with a model of its structure and behavior. Also, the system control software is defined independently of the configuration, using generic planning and scheduling algorithms based on a real-time constraint solver library. Thus, the task of software configuration becomes one of composing the module models to system models and deriving the system capabilities that arise from the module capabilities (a planning step). As mentioned, since the original CDL models have a dual CCP representation, we can use CCP techniques for this reasoning process. In fact, the primary technique used is partial evaluation of CC programs.

The process of composing module models to system models is complicated by a number of problem and modeling properties. First, modules are modeled as endless CC processes, and the system configuration may contain physical loops (e.g., a print engine’s duplex loop for double-sided printing). Thus, capability derivation may result in incomplete or non-minimal system capabilities, or not even terminate. (A system capability is complete if, for each selected module capability, the inputs or outputs are present at the model’s interface. As mentioned, since the original CDL models have a dual CCP representation, we can use CCP techniques for this reasoning process. In fact, the primary technique used is partial evaluation of CC programs.

The process of composing module models to system models is complicated by a number of problem and modeling properties. First, modules are modeled as endless CC processes, and the system configuration may contain physical loops (e.g., a print engine’s duplex loop for double-sided printing). Thus, capability derivation may result in incomplete or non-minimal system capabilities, or not even terminate. (A system capability is complete if, for each selected module capability, the inputs or outputs are present at the model’s interface. As mentioned, since the original CDL models have a dual CCP representation, we can use CCP techniques for this reasoning process. In fact, the primary technique used is partial evaluation of CC programs.

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 CDL-to-CCP compiler annotates conditions with the heuristic. Such annotations allow us to control the abduction of other “types” of conditions (e.g., to control recursion). Thus, such annotations enable domain-specific reasoning within a domain-independent modeling language.

Conclusion

Our approach to developing software for complex systems is part of a larger vision for model-based computing: to support both human communication and computer processing through formal representations. To that effect, we provide a domain-specific language to the engineer, but define its semantics and reasoners in terms of the generic CCP framework (Fromherz, Gupta, & Saraswat 1997). Reasoning for software configuration then consists primarily of the composition and partial evaluation of CC programs, making use of compiler annotations where domain-specific heuristics are needed.

Acknowledgments

The work reported here has been carried out in collaboration with a number of past and present researchers and engineers within Xerox Corporation, in particular Vijay Saraswat, Danny Bobrow, and Vineet Gupta.
References

Appendix
A. CCP Syntax
Our CC programs are defined by clauses $H :: S$ with head $H$ and body $S$. The abstract syntax of a CC program is defined as follows.

Program $P ::= H :: S \mid P.P$ (Head, body)

Statements $S ::= c$ (Tell)

$[A] \mid [A; S]$ (Conditional)

$S, S$ (Conjunction)

$x^* S$ (Local variable)

$H$ (Process call)

$A; A$ (Alternatives)

Condition $C ::= C \rightarrow NS$ (Ask-tell)

$A; A$ (Alternatives)

Annotation $N ::= c \mid c; c$ (Constraints)

Process head $H ::= p(x_1, \ldots, x_n)$ (Name, args)

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 $p$.

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 (annotated) conditional $[C_1 \rightarrow N_1, S_1; C_2 \rightarrow N_2, S_2; \ldots; S]$ is written as

if $C_1$ abduction $N_1$ then $S_1$

else $C_2$ abduction $N_2$ then $S_2$ ...

else $S$

The syntax of constraints $c$ depends on the constraint system. 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 $L = [H | T]$.

B. CCP Semantics
Computation of CC programs progresses by monotonically accumulating constraints in the store, and by checking whether the store entails constraints. Synchronization between processes is specified by the ask construct: a conditional $[C_1 \rightarrow N_1, S_1; C_2 \rightarrow N_2, S_2; \ldots; S]$ 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 $S$. 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. The semantics of a CC program is defined as follows.

Tell $(\langle S, c \rangle, s) \rightarrow (\langle S, s \cup c \rangle)$

Ask if $\langle S, [A; C \rightarrow . S] \rangle, s \rightarrow (\langle S, S \rangle, s)$

Ask else $\forall C. (C \rightarrow _{c, \ldots, c}) \in A \rightarrow s \cup C \uparrow false$

$\langle (S, [A; S]), s \rangle \rightarrow (\langle S, S \rangle, s)$

$\langle (S, A; C \rightarrow N[S]), s \rangle \rightarrow (\langle S, S \rangle, s)$

$s \uparrow C \uparrow false$

Ask abduction $\langle (S, [A; C \rightarrow N[S]], s) \rightarrow (\langle S, s \cup C \rangle)$

Variable substitution $\langle (S, x^* S), s \rangle \rightarrow (\langle S, S[y/x] \rangle, s)$

$\langle (S, y \notin \text{var}(S)) \rangle$

Process call $h :: S$ a clause in the program

$(\langle S, h \rangle, s) \rightarrow (\langle S, S \rangle, s)$

$(S, S)$ stands for selecting $S$ anywhere in a conjunctive statement, with $S$ being the (potentially empty) rest of that statement, and $\{A; A\}$ stands for selecting $A$ anywhere in a conditional statement, with $A$ being the (potentially empty) rest of that statement.