Template Design of Discrete-Event Systems

Technical Report No. 2007-538

Lenko Grigorov

grigorov@cs.queensu.ca
School of Computing
Queen's University, Kingston, Canada

Abstract

A new methodology for the design of DES control is proposed which allows for the creation of conceptual designs by using encapsulated low-level elements. The approach can be used within the standard framework of modular supervisory control. The notion of *DES templates* is introduced, where both DES modules and specifications are represented as high-level abstractions of typical behaviors. The control engineer creates instances of these abstractions and specifies the way the instances interact. System modeling and the design of specifications occur simultaneously. Rapid prototyping with implicitly consistent models improves the speed and robustness of the design process. The implementation of software for template design is described, as well as the application of the proposed methodology to a small system to get real-world feedback.

August 8, 2007

Chapter 1

Introduction

In control engineering, the possible behavior of some systems can be described as a set of sequence of discrete events (events that occur instantaneously). Ramadge and Wonham [14] propose a theoretical framework, called Supervisory Control Theory, for the modeling and control of such systems. In this framework, the discrete events are instantaneous, spontaneous, and certain control can be exercised by preemptively blocking some of them. Systems modeled in this framework are called Discrete-Event Systems (DESs) and the entity exercising the control is called a *supervisor*.

In order to obtain a supervisor for a system, the control engineer needs to design the system as a DES, provide the control specifications in the form of an "ideal" DES, and invoke a special computational algorithm that computes the most permissive supervisor which satisfies all restrictions.

In theory this approach works well, however, practical implementations have run into a number of problems. The most significant problem is what is called "state-space explosion". The state complexity of a system model may grow exponentially with the number of participating subsystems. Another problem for the use of the theory in practice is the fact that modeling a system and verifying the end result is difficult and non-transparent for the users [19]. Further complications arise from the fact that the usability of software packages for DES control is generally unsatisfactory and that generally there is no support for the use of an abstract supervisor in the control of a real system.

The problem of state-space explosion has been addressed partially by considering *modular*, [20], or *hierarchical* supervision, [13], and by dealing with systems incrementally, [1]. Of these methods, modular supervision seems to be most mature. The system is modeled as a set of separate modules or subsystems which may interact. Usually, control specifications can then be given in a modular fashion as well—concerning only a subset of all the modules. The reduction of complexity is a result of being able to compute separate, smaller, supervisors for each separate specification. Incremental approaches to DES control usually also rely on having a modular system model. Then, compositions of modules are constructed only as needed in order to determine a given property of the system. In hierarchical control, the base system is usually abstracted in a specific fashion and then supervisors can be computed just for the simpler high level of the system. Unfortunately, the research done on hierarchical supervision is more disparate and a unifying theme is lacking [9]. Modular control is not

without problems either. When separate supervisors are constructed for each specification, it is not possible to predict what will be the net effect of the simultaneous application of all supervisors. Sometimes, due to some interdependency between the different control policies, the system may block. Thus, after the separate supervisors are constructed, it is necessary to check if the simultaneous application of these supervisors will lead to blocking. For this purpose, all supervisors have to be composed, which in some sense forfeits the benefit that is achieved by constructing separate supervisors. However, since blocking is a global property, in the general case there is no way to avoid the global check.

During our personal experience with the application of supervisory control, we noted further complications. First, control engineers are required to learn about Finite-State Automata (FSAs) and the modeling of systems and specifications using FSAs. The process of modeling is quite slow and requires a lot of attention in order to avoid errors. The presence of an error in the design is not readily observable. The design of interaction between different system modules is achieved through synchronization on common events. The designer needs to constantly maintain an overview of what events are used for the purpose. A second complication, partially a result of the method for synchronization, is that system models are usually designed on a per-use basis. It not trivial to reuse models in other projects. Many times it is necessary to start modeling from scratch even though parts of the model have been designed on previous occasions. Third, the application of the algorithm for supervisor generation frequently results in an huge entity which cannot be comprehended completely by humans; in moderate-size projects a supervisor may have hundreds of states and thousands of transitions. This necessitates some other method of testing the functionality and the performance of the output, e.g., by simulations or by test runs. This, in turn, requires that the abstract supervisor obtained at the end of the modeling process be implemented in some concrete form. Most popular DES software, such as CTCT [2], DESUMA [6] and others, does not offer any tools for automated supervisor implementation.

While there does not seem to be an easy solution to this complex set of issues, the use of predefined DES units by engineers may lead to a much easier application of supervisory control. In [7], the authors describe an approach where the controlled behavior of a discrete-event system is designed using a set of very simple specifications. Each specification is built from a prototype structure, a *template*, and exercises control over a single aspect of the system—such as the operation of a gripper. All specifications are executed in parallel and thus, simultaneously, provide control for the whole system. The benefits pointed out by the authors include significant reduction of the time needed to design controllers (e.g., one hour versus 12 hours), lower cost of the project (the approach encourages the substitution of software complexity with cheap hardware sensors) and more robust handling of failures (no need for complex reset procedures). However, this approach also has some disadvantages. It is assumed that almost all system behavior can be described as the concurrent execution of simple units without much interaction. This is not suitable for the definition of global specifications, such as the control for non-blocking. The suggested templates seem too simple to express more complex requirements. Furthermore, the methodology is not cast within the supervisory control framework and it cannot take advantage of the algorithms therein.

In this work, we propose a new methodology for the design of DES control. We introduce

the notion of *DES templates* within the framework of supervisory control. Both DES modules and specifications are represented as high-level abstractions of typical behaviors. The control engineer creates instances of these abstractions and then needs only to specify the way the instances interact. System modeling and the design of specifications occur simultaneously. Speed and robustness of the design process are improved since it is not necessary to deal with details of the system behavior, as well as to reimplement similar parts of a system. The computation of the supervisory solution can be automated. The methodology was implemented in software and support for the generation of Programmable Logic Controller (PLC) code was added. Then, we applied the approach to obtain a control solution for the hardware of a small system.

Next, we describe our approach and give a brief motivation for each aspect. Then, we discuss the software implementation. Finally, we discuss our observations when the approach was applied for the control of a robotic testbed.

Chapter 2

Template design of DESs

The template method for DES design is substantiated greatly by the observations made during a study of how humans solve DES control problems [10]. When faced with a new problem, subjects frequently engaged in drawing a simple diagram of interactions between parts of the system which needed to be modeled. It appeared that the subjects liked to isolate different aspects of a system before they proceeded with the low-level modeling. These observations led to the proposal for a new methodology for the design of DESs, where control engineers can focus on assembling together blocks of subsystems and specifications instead of worrying about every little detail of the system.

2.1 Framework

The framework for template design is largely based on the work of Santos *et al.* [17, 16]. The authors propose a methodology for conceptual design of DES system using *instances* and *channels*. Instances are the active parts of the system (e.g., workstations). Channels are passive parts of the system which facilitate the transfer of matter and energy between instances (e.g., conveyor belts). It is argued that this framework is suitable for the modeling of complex systems since it allows the simultaneous definition of both structure and functionality.

When the methodology of [17] is applied to discrete-event systems, in effect modular control is used where DES modules become instances and specifications serve as channels. Then, the supervisors for the system are computed according to the local modular control proposed in [4].

In our framework we decided to keep all the basic propositions of [17], however, we decided to cast the whole idea purely in DES terms. Thus, a system model consists of a set of modules (DES subsystems), M, a set of channels (DES specifications), N, and a function defining the links between modules and channels, C. Modules and channels are modeled as finite-state automata (FSAs), $G_i = (\Sigma_i, Q_i, \delta_i, q_{0i}, Q_{mi})$. Furthermore, all modules and channels have to be asynchronous, i.e.,

$$\forall i \neq j, G_i, G_j \in M : \Sigma_i \cap \Sigma_j = \emptyset$$

$$\forall i \neq j, G_i, G_j \in N : \Sigma_i \cap \Sigma_j = \emptyset$$

$$\forall G_i \in M, G_j \in N : \Sigma_i \cap \Sigma_j = \emptyset.$$

The requirement that modules be asynchronous is not a stringent restriction as discussed in [4]. The benefit of having asynchronous modules is mainly in being able to make more uniform assumptions about the system. If some modules are not asynchronous, they can be composed until there are no dependencies between modules. The channels have to be asynchronous because they describe only some abstract specifications which still need to be concretized in terms of the given system. This concretization is done with the help of links. Let $\Sigma_M = \bigcup_{G_i \in M} \Sigma_i$ and $\Sigma_N = \bigcup_{G_i \in N} \Sigma_i$. Then $C : \Sigma_N \to \Sigma_M$. In other words, the function defines links between events of modules and events of channels. The interpretation of the link $C(\tau) = \sigma$ is that the event τ in the given channel should be considered equivalent to the event σ of the given module—thus relating the abstract specification to the given system. This allows the synchronization of modules through the channels, in effect defining the protocols for the transfer of information between parts of the system.

After a system is modeled in the proposed framework, modular control can be applied to obtain supervisors for the separate specifications. In our work we propose the use of an optimized version, namely local modular control [4]. The precondition for the application of this method is satisfied, i.e., the participating modules are asynchronous. All modules which are linked to a channel participate in the subsystem influenced by the specification determined by the channel. Let $G_{channel}$ be a channel. Then

$$C(G_{channel}) = \{G_i | \exists \tau \in \Sigma_{channel}, \exists \sigma \in \Sigma_i, C(\tau) = \sigma\}$$

is the set of modules influenced by $G_{channel}$. For every channel $G_{channel}$, all the modules influenced by it are composed via synchronous product.

$$G_{subsys} = ||_{C(G_{channel})} G_i$$

Then all events in the subsystem which do not appear in the channel are applied as self-loops to all states in the channel, i.e., the channel has no influence on the occurrence of these events.

$$G_{spec} = \text{selfloop}(G_{channel}, \Sigma_{subsys} \setminus \Sigma_{channel})$$

Finally, the common algorithm for the construction of the supremal controllable sublanguage of the channel with respect to the relevant subsystem is invoked.

$$\text{supcon}(G_{subsys}, G_{spec})$$

As a result, local supervisors for each channel are constructed. The last step involves checking the mutual non-blocking of the set of supervisors, as described in [4].

2.2 Templates

The next advantage of our methodology is that it allows the use of templates. A template is simply a model of some discrete-event behavior. In the supervisory control setting, the model would be an FSA. In other words, any FSA can be a template. The idea behind

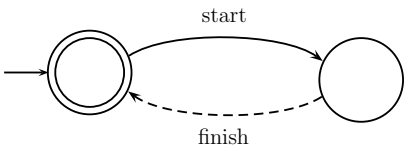


Figure 2.1: The behavior of a typical workstation.

templates is that if they define some frequently used behavior—such as "workstation" (see Fig. 2.1)—one need not manually create a separate FSA each time this behavior is needed. Instead, the software can make a copy of the template, or *instantiate* the template. Thus, for example, creating the DES modules for ten workstations would be reduced to instantiating a template ten times. Since the copy is made automatically, the process is both faster and less error-prone. Furthermore, if the templates have been designed by experts and thoroughly tested, any user can use them with the same degree of reliability.

Since templates can describe both system behavior (i.e., modules) and restrictions on behavior (i.e., channels), the use of templates within our framework is very natural. Suppose there is a library of templates (FSAs). Then, the set of modules, M, participating in a design can be created by instantiating the required templates. Since the events of every template instance are named in a unique way, all modules will be asynchronous as required. Similarly, the set of channels, N, can be created by instantiating templates.

In order to simplify the creation of links between modules and channels, the definition of a template may indicate a set of *interface* events $\Sigma_f \subseteq \Sigma$. If some of the events of a module describe only internal functionality and are not relevant to other modules, then they will not participate in synchronization links. The events that may be used for synchronization are chosen to be the interface events of a template. Only interface events will be available to the designer to create links between modules and channels. Since channels have no inherent functionality, all events are supposed to be used for synchronization. Thus, channel templates by definition will have only interface events.

It is also important to note that if templates are used for the design of a DES, the control engineer need not know all the details of FSA modeling. If the necessary templates are available, it is possible to design the system completely at the abstract level of template instances and links between them. It is sufficient to understand what behavior is described by each template and to understand what the interface events stand for.

2.3 Parametrization

A further improvement to the template design methodology can be brought by considering parametrization of the template behavior. For example, if one would like to create templates for buffers, a separate template has to be constructed for all buffer capacities that need to be considered (e.g., buffer with two slots, buffer with three slots, etc.) However, it can be easily seen that the basic workings of a buffer are the same regardless of capacity. It would be much more convenient if there were a single "buffer" template which is parametrized in terms of

capacity—and then at instantiation one would be able to choose the specific capacity to be used.

One possible approach to parametrization of FSAs is described in [3]. There, a regular FSA is augmented with a *data collection*. The data collection is a vector of scalars which can range over some set, e.g., the set of integers. A vector of unary functions is associated with each transition in the FSA. When the transition occurs, each function in the vector is applied to the corresponding scalar in the data collection. For example, a buffer can be modeled as a single state with two self-looped transitions, "insert" and "remove" for inserting elements to the buffer and removing from it, respectively. The number of items in the buffer will be recorded with the help of a data collection with a single integer. When the "insert" transition occurs, the function "+1" will be applied to the integer. When the "remove" transition occurs, the function "−1" will be applied. The control of such a discrete-event system may take advantage of the data collection in addition to the information in the states of the FSA. Indeed, control can be based on predicates about the current state of the system and on the current value of the data collection. Then, the authors propose a method to compute the supremal controllable sublanguage of a system by incrementally backtracking with the predicates until the control decisions do not attempt control of uncontrollable events.

Among the advantages of this type of model is the fact that it is possible to express non-regular behaviors and specifications. Unfortunately, this is also the reason why the model cannot be readily applied in the template framework proposed in this work. A potential solution would be to restrict the type of data collections that can be used. For example, if each scalar in a data collection is restricted to belong to a closed integer interval, then the behavior of the system will become regular-only. However, even in this case it is not possible to use this type of model directly. It is necessary to find an efficient transformation from the parametrized model into a "simple" FSA.

Another approach to template parametrization is a method where a certain subset of states in an FSA is multiplied and chained a given number of times, depending on an integer parameter (see Fig. 2.2). This approach will make it easy for experts to graphically design templates which are parametrized. Furthermore, for any specific value of the parameter, a "simple" FSA is obtained. It is not necessary to perform any conversions before an instance of the template is used in the design. However, we have not yet investigated all aspects in detail and it is unclear how exactly the parametrization will function in a complex FSA. More work needs to be done before this method becomes viable.

2.4 Benefits

The template design of DESs is based on theoretical work, however, the main motivation for its conception was making the application of DES control simpler. It is expected that using template design will result in the following improvements.

- Faster design of systems. The use of pre-built templates not only reduces the time to mechanically input new FSAs but also the time to mentally consider low-level details of FSA implementations.

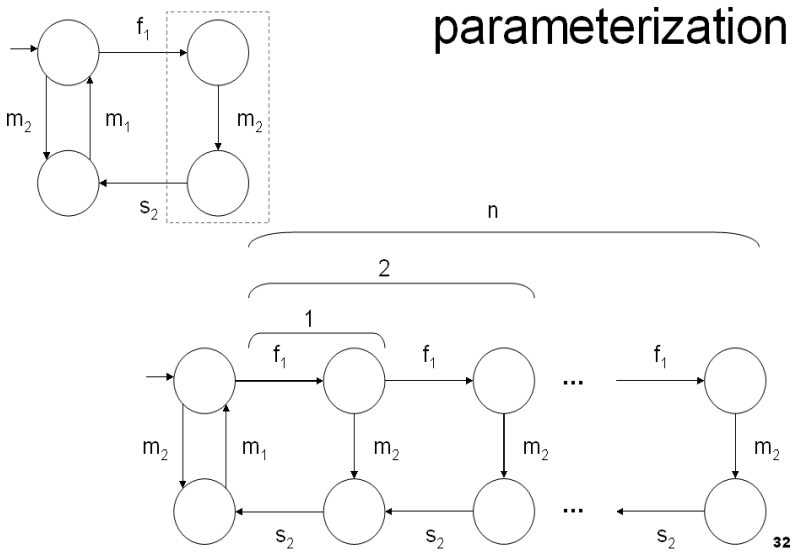


Figure 2.2: Graphical approach to parametrization.

- More robust designs. Fewer errors can be made during the design since it is not necessary to manually copy FSAs and to keep track of the names of events in different modules and specifications. All events are by default unique, and synchronizing modules simply requires the establishment of a link.

- Easier design. The designer need not keep track of the FSAs which underly every template or template instance. The creative effort can focus only on the important task of determining which modules and channels are to be used and how to link them. The creation of a supervisor is completely automated.

In order to see if these expectations are substantiated, it is important to actually apply the methodology in practice. Thus, we decided to build software which supports the proposed method. Then, we used the software in order to design a controller for a small robotic testbed.

Chapter 3

Software package for template design

The theoretical foundation of template design does not require any graphical representations. However, as it has been discovered in practice [8, 10], the lack of graphical representation may significantly influence the usability of a software package. Thus, one of the principles we decided to follow in designing our software was to use a graphical interface. A prototype of the interface is shown in Fig. 3.1. We decided to use boxes to represent instances of modules and circles to represent instances of channels. The links between modules and channels are represented as lines connecting the boxes and circles. The user of the software can create and manipulate the graphical elements using the mouse cursor. As well, the mouse is used to establish the links between modules and channels. The choice of a predominantly mouse-based interface was made due to research showing that working with the mouse is generally faster than using keyboard shortcuts and commands [18].

Template design is just an abstract interpretation of modular supervisory control. Finite-State Automata underlie all elements of the design and regular DES operations are applied at the low level to produce the supervisors. Thus, any software which supports template design has to be able to perform all functions of a regular DES tool as well. Instead of writing a completely new software package, we decided to extend existing DES software to add the template design functionality. The IDES software developed at Rudie's research laboratory, [11, 15], was chosen since its architecture supports the addition of extensions, it offers an advanced graphical interface infrastructure, and we have access to the source code. Furthermore, IDES is implemented in the Java programming language, so it can be used on all major computer platforms.

3.1 PLC code generation

Since the purpose of template design is to make the application of DES theory easier, we decided to try to streamline the complete process of application: from modeling to control of the real hardware. We envisioned that a control engineer will be able to model a system using template design and then they will be able to automatically obtain the control code for the real system.

In many cases the real system is controlled by a PLC unit; this is the case in our ex-

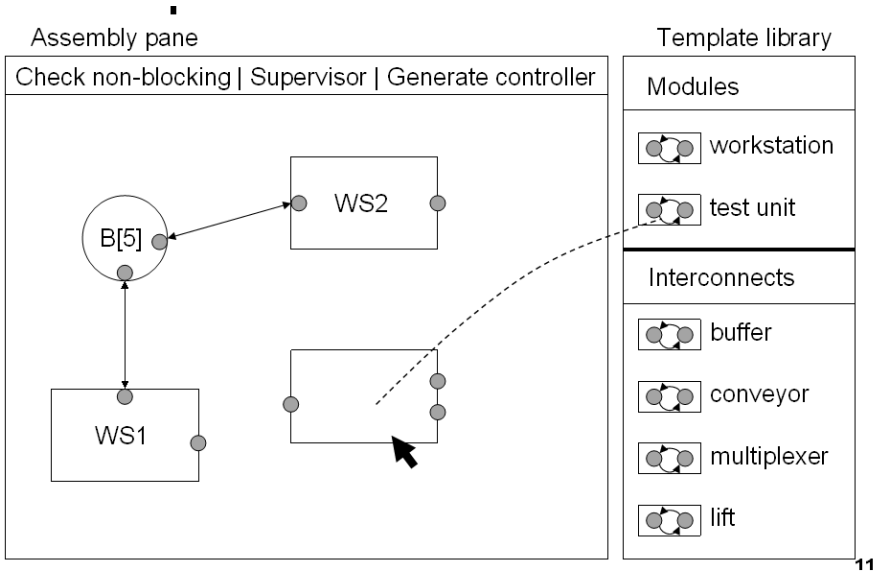


Figure 3.1: Prototype of the template design software interface.

ample system as well. Thus, we decided to focus on the generation of PLC code from the template design. However, it is possible to generate other types of code, e.g., C code for microcontrollers, as shown in [21].

The solution of a supervisory control problem is in the form of supervisors modeled as FSAs. Usually, the events in such a system are abstractions of sequences of low-level events that may occur in the real system. For example, the event "move robot arm to location 2" may involve a sequence of electric signals sent to one of the motors in order to rotate the needed amount. There is a gap that needs to be filled between the abstract controller represented by the DES supervisors and the low-level controller for the real system. The first step is to convert the supervisor logic into control code.

There are many ways how to convert FSAs into code, however, the method proposed in [5] seems to be most suitable for two reasons: it converts FSAs directly into PLC code, and it is designed with modular control in mind. The authors propose a method where the transitions of the supervisors are implemented in PLC code and the control data for each state is available. The transitions of the supervisors are triggered by signals from another part of PLC code, namely, the DES system layer. The FSAs of all the system modules are converted into PLC code similar to the code for the supervisors, thus forming the DES system layer. This layer in turn communicates with the real system: inputs from the hardware signify the occurrences of uncontrollable events, while the controllable events force the execution of subroutines for the system. The DES system layer is also responsible for sequencing and prioritizing events which happen concurrently.

The method for PLC code generation described so far relies on manual modifications to the generated code in order to insert hardware-specific instructions. For example, the generated code needs to call subroutines for the controllable events. When the code is generated, it

is assumed that the code for such subroutines will be provided manually. Full automation is not achieved. We propose a way to bridge this last gap in the following way. For each event in the template design, the user can specify a snippet of PLC code. In other words, the low-level subroutine for each event can be specified within the design. Then, during PLC code generation, this subroutine can be inserted as needed.

The fundamental issue which needed to be resolved was how to couple the automatically generated PLC code for the supervisors and the DES system with the code snippets entered by the user. The generated code cannot be aware of the code entered by the user (the custom code can be in any form and can use any variables), nor can the user be aware of the generated code in advance (it is not available until the system modeling is finished). Thus, it was necessary to establish a protocol of communication between the two parts. The protocol was chosen to be as follows. The user may specify a range of memory which will be used by the custom code and the generated code will not use. Furthermore, the generated code will communicate with the code snippets through special variables for each event. In the case of uncontrollable events, when the snippet sets the corresponding variable to "true", it will announce that the event has occurred. In the case of controllable events, when the generated code sets the corresponding variable to "true", it will request that the subroutine for this event be executed. In the custom code snippets, these binding variables can be referred to by using a special placeholder string. When the complete PLC code is generated and the memory location of the variables is known, the placeholder strings will be replaced with the real variables.

3.2 User interface

The user interface which was implemented follows closely the interface prototype of 3.1. A snapshot of a template design opened in IDES can be seen in Fig. 3.2. The interface is not polished yet and many things need to be improved until it becomes fully usable. However, all major components are functional and it is possible to accomplish the following tasks:

- Create a template from an FSA. The template can be either a module or a channel. In the case of a module, it is possible to select which events will be interface events.
- Open a template from the template library to explore the underlying FSA.
- Remove a template from the template library.
- Instantiate a template. The instance automatically receives a unique name and all events are prefixed with this name. As a result, all instances in a design are asynchronous by default.
- Open an instance to explore the underlying FSA.
- Remove an instance from the design. All corresponding links are automatically removed as well.
- Rename an instance. All events of the instance are automatically renamed as well.

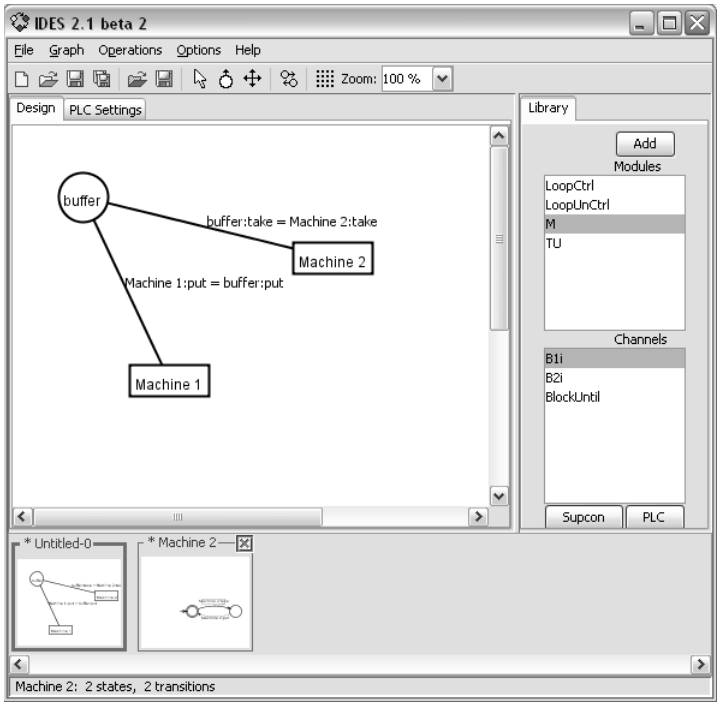


Figure 3.2: The template design software interface in IDES.

- Create a link between an instance and a channel. The equivalence of the events will be displayed graphically beside the link.
- Remove a link from the design.
- Compute the modular supervisors for all channels. The local modularity, [4], of the supervisors is also checked.
- Enter the PLC code for the events of the modules.
- Generate the PLC code for the control of the hardware system, using the modular supervisors and the PLC code for all events.
- Store a design to the disk and load it back.

The representation and manipulation of a template design is predominantly graphical and mouse-driven. When the user decides to work with the underlying FSAs, IDES switches to the regular FSA mode of operation and all regular tools are available to the user.

3.3 Implementation

The software for template design of DESs was implemented as an extension of IDES. In order to extend it, the following main components were developed:

- *TemplateModel*: An implementation of the theoretical model of template designs. This corresponds roughly to the implementation of the theoretical model of an FSA (such as states, events, and transitions).
- *TemplateDesign*: A shell which is applied to a TemplateModel and stores all relevant graphical layout information. This corresponds roughly to the component responsible for managing the layout information of an FSA graph (such as nodes, labels, edges).
- *Design(Drawing)View*: The interface component which renders a template design on the screen and lets the user interact with it (such as moving a template instance with the mouse).
- *TemplateIO*: XML file input/output for template designs.
- *LocalModularity*: A DES operation to check the local modularity of supervisors.
- *ConstructSupervisors*: A DES operation which constructs the modular supervisors for all channels in a template design. Since regular DES operations such as "meet", "self-loop" and "supcon" are used, first it is necessary to rename the events in linked modules and channels to synchronize them properly. This is done completely automatically.
- *TemplateLibrary*: A collection of module and channel templates which are available for instantiation. The library supports the addition and removal of templates and provides automatic persistence (storage of the templates on the disk).

In order to implement PLC code generation, we integrated the BAJ library co-developed together with Francisco da Silva at the Federal University of Santa Catarina, Brazil. Given a set of FSA modules and FSA supervisors, this library generates the control code in the Instruction List PLC language, compatible with Siemens PLC units. Our software then appends the code snippets for all events to this output. The placeholder string, "<event>", in the snippets is replaced since at this time the name for all binding variables are known. The PLC code is not compiled since a stand-alone compiler for Siemens is not available. The user has to manually open the code in the software provided by Siemens and then compile it and download it into the PLC unit.

3.4 Improvements necessary

While the software developed is functional and it was successfully used in a small project, it is still in a very preliminary form. In order to become as usable as the FSA manipulation part of IDES, it is necessary to complete and polish the implementation of many components. The things that need to be improved include:

- Improvement of the template library interface. Most UI elements break design recommendation either with form or behavior or both. For example, the "remove template" functionality is hidden—the user needs to press the "Delete" button on the keyboard but this is not obvious from the interface.

- Improvement of the presentation of a template design. Most of the elements are ugly. For example, it may be better to use ovals instead of circles to represent channels. Furthermore, links are straight lines between the centers of the boxes and circles of the corresponding modules and channels. It would be much nicer to use broken lines as used in schematic software. The algorithm for placement of link labels is very simple and usually the results are unsatisfactory.

- Improvement of the method of interaction with the user. There are many glitches in the interface which contribute to an overall unsatisfactory experience for the user. For example, in order to instantiate a template, it is necessary to first click on the name of the template in the library, and then click again in order to drag it onto the design canvas. Also, some interactions with the instances on the canvas are contrived—such as having to select "Delete" from a pop-up menu instead of just pressing the "Delete" button on the keyboard. As well, if there are multiple links between a module and a channel, it is not possible to select a specific link for manipulation or deletion.

- Improved consistency of the template design. There are no checks for user inputs which may result in violations of the model constraints. The user may name two template instances with the same name (thus, in effect, creating identical, perfectly synchronized, copies). As well, the user may establish a link between two modules or two channels, which is meaningless within the proposed framework.

- Improvement of the I/O system for template designs. The I/O support is currently very rudimentary and fragile. For example, IDES knows if the model one is trying to load is a template design only from the file name, i.e., if it is "TemplateDesign.xmd", the model is assumed to be a template design, otherwise it is loaded as an FSA. The storage of the template design into an XML file has many assumptions about the form of the stored PLC code which may be not be always true.

Chapter 4

Example application

In order to test the applicability of template design of DESs, the methodology was used to design a controller for a robotic testbed at the Department of Automation and Systems, Federal University of Santa Catarina, Brazil. A photograph of the hardware is shown in Fig. 4.1. The intended functionality of the system is to retrieve parts from an input buffer, perform operations on the parts and test if the operations were successful. If so, the given part is output into an "accepted" buffer. If not, the part is placed into a "reprocess" output buffer. From there, a part can be manually moved into a "reprocess" input buffer. The system will reprocess parts from this input and if the operations are unsuccessful again, the parts are output into a "rejected" buffer. The subsystems in the testbed include:

- A set of plastic parts with or without a metal screw;
- Input buffer for new parts, with a sensor;
- Input buffer for parts to be reprocessed, with a sensor;
- Output buffer for successfully processed parts;
- Output buffer for parts which need to be reprocessed;
- Output buffer for rejected parts;
- A rotating table with four slots. It has a positional sensor for calibration;
- Robotic arm with a grabber. The arm has 360 degrees of freedom of rotation, and a positional sensor for calibration. It can reach all input and output buffers, as well as one of the slots of the table. The grabber can close to retrieve a piece from the input buffers or the table and open to deliver a piece into the output buffers or the table. It has a positional sensor for calibration;
- "Drill" workstation with a sensor for the presence of a part;
- "Welder" workstation with a sensor for the presence of a part; and

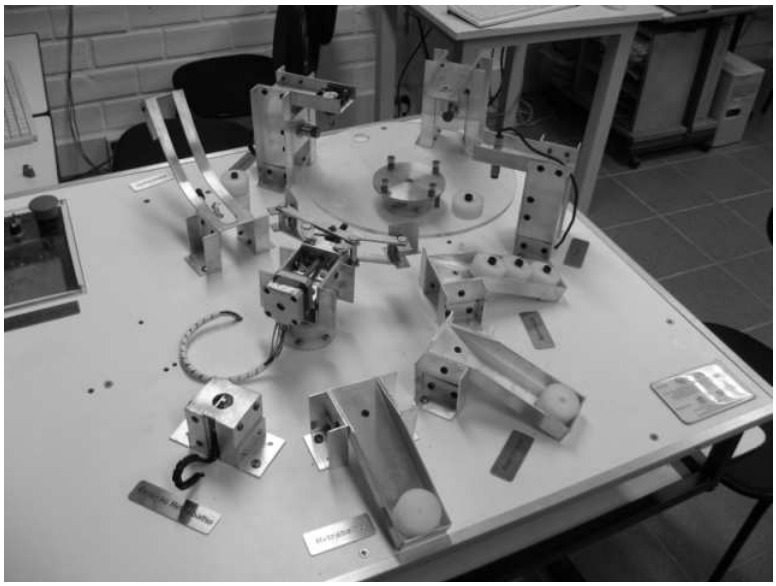


Figure 4.1: The robotic testbed where template design was applied.

- Test unit with a sensor for the detection of the success of the operations (if the part has a screw).

The system is controlled via a Siemens S7-200 series PLC unit, connected via Profibus to a PC. The PLC code is compiled and downloaded onto the PLC using the Step 7 Micro/WIN software package provided by Siemens.

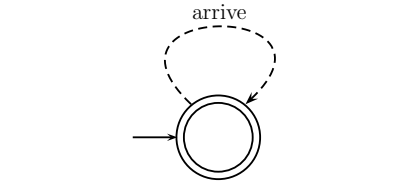
4.1 System model

The model of the system had to be greatly simplified in order to be able to complete the project within the constraints listed below:

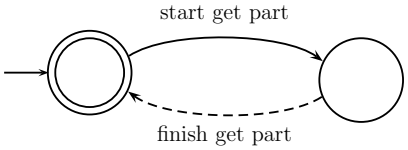
- Time—the research group had a very limited window of opportunity to develop and test the supervisory control solution.
- PLC memory size—the PLC disposes of only 4 kilobytes of memory.
- Software limitations—the IDES software used is still under development and cannot handle very big FSA models.

Further investigation done by Klinge, [12], shows that using some workarounds and supervisor reduction techniques, it may be possible to obtain a supervisory solution for a much more complex model.

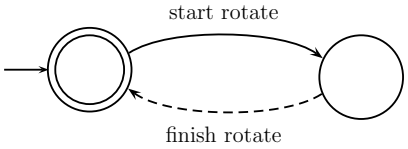
The part of the system we used included four modules: the input buffer for new parts, the arm with the grabber, the rotating table and the drill. The arm with the grabber was



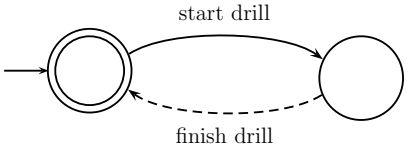
(a) Input buffer



(b) Arm with grabber



(c) Rotating table

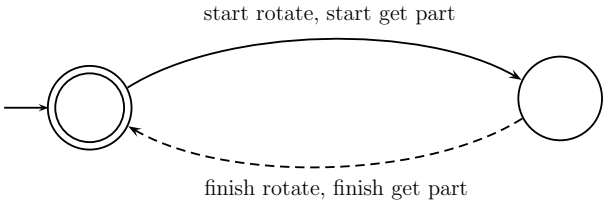


(d) Drill

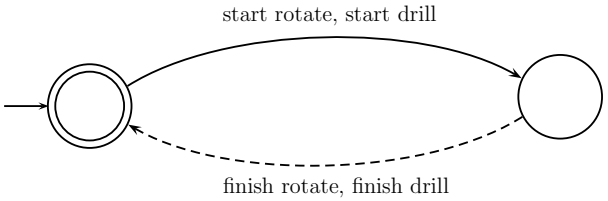
Figure 4.2: Module models.

simplified to perform only one (high-level) activity: retrieve a part from the input buffer and place it on the table. The FSA models for all modules are shown in Fig. 4.2.

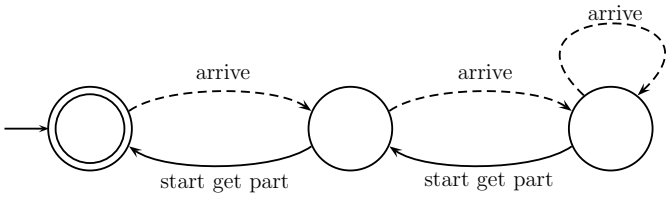
The specifications applied to the system were as follows. First, there has to be mutual exclusion between the table and the arm and between the table and the drill. In other words, the table should not turn while one of the other units is in the middle of completing an operation. Second, there has to be underflow control for the input buffer, i.e., the arm should not retrieve a part if there are no parts in the buffer. The buffer is modeled as having two slots but without overflow control since physically there cannot be more than two parts available at a time. Third, there has to be control over the sequence of operations: after a part is placed on the table, the table has to turn before the drill operates on the part. Since turning the table frees up the slot where parts are deposited, a second part can be placed before the drill operates. In order to keep the model simple, there is no control on how many parts are located on the table, i.e., the system may attempt to process five parts even though after the fourth part, the table is full (parts are never removed). The FSA models for all specifications (or channels) are shown in Fig. 4.3.



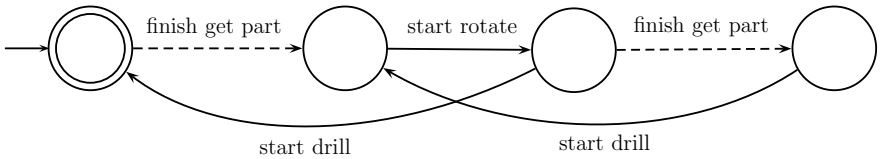
(a) Mutual exclusion between table and arm



(b) Mutual exclusion between table and drill



(c) Underflow control for input

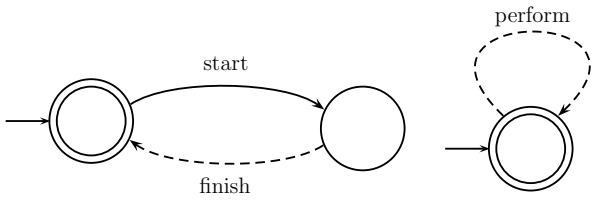


(d) Operation sequencing

Figure 4.3: Specification models.

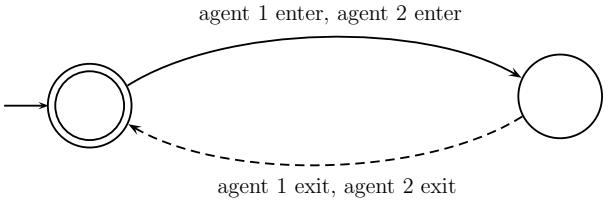
At the start of the modeling, it was assumed that the template library contains all relevant templates (see Fig. 4.4). Then, the modules were created by instantiating the "Workstation" template for the arm, table and drill and by instantiating the "Selfloop" template for the input buffer. The channels were created by instantiating the "MutEx" template twice and the "Buffer2slots" and "Sequence" templates once. Afterwards, the relevant events were linked. For example, the "perform" event of the part input is linked with the "in" event of the buffer channel. After the model is ready, the PLC code for each event in the system is manually input. For example, the code for the event "start drill" is

```
Network 1
LD     <event>
S      M3.6, 1
Network 2
LDN    M3.6
```

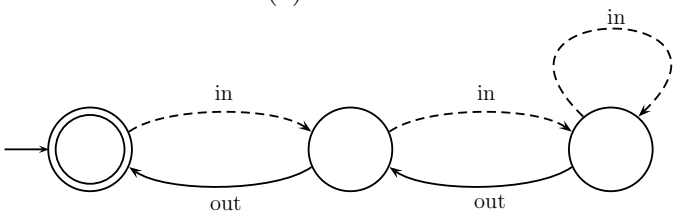


(a) Workstation

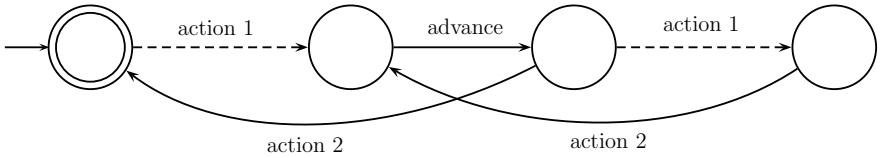
(b) Selfloop



(c) MutEx



(d) Buffer2slots



(d) Sequence

Figure 4.4: Template models.

```
JMP    11
Network 3
LD     M3.6
TON    T102, +50
S      Q0.4, 1
Network 4
LD     T102
R      M3.6, 1
S      M3.7, 1
R      Q0.4, 1
R      T102, 1
Network 5
LBL    11
```

After the template design is ready and the PLC code is input, the user presses the "PLC" button to obtain the final solution. The software computes the modular supervisors for all channels, checks if they are locally modular, and then generates the complete PLC program for the control of the system, including the low-level subroutines for the abstract events in the model. An additional output, as required by the Siemens control software, is a file containing the mapping between variable names and memory locations.

In order to complete the application, and start up the hardware system, the user needs to launch the Siemens Step 7 software, copy the variable mappings from the generated file and then import the PLC code. Compiling and downloading to the PLC unit is done by pressing the corresponding buttons in the Siemens software.

When we tried this approach, we had to use "virtual" parts in the testbed, since the low-level code for the control of the grabber still had unresolved issues. The grabber was not capable of removing a part from the input buffer. Thus, we had to activate the input sensor by hand—and then the system proceeded as if it were operating on a real part. Furthermore, we discovered another issue with the low-level code for the handling of the arrival of new parts. The event signaling an arrival was generated depending on the level of the sensor signal. Since the cycle of the PLC unit takes just a fraction of the time for which the signal is "on", this resulted in an unbounded number of arrival events generated for each activation of the input sensor. Since the buffer channel in the model has only two slots, however, the controller received only two arrival events per sensor activation.

Overall, the application of the proposed methodology was successful, resulting in very minimal time for the modeling and deployment of PLC controllers designed through supervisory control.

Chapter 5

Observations

The application of the template design methodology to a real project, even though very small, brought some interesting insights.

Surprisingly, the biggest advantage of the design methodology does not seem to be the ability to use templates. Even in the small example project, it was necessary to use custom, project-specific modules and channels which would not make very reusable additions to the template library. However, the users' experiences were quite positive in other respects. First, the template design environment made it very easy to model and remodel systems, i.e., to create prototypes in the initial stages of system design. It is simple to replace modules and channels and then generate the corresponding supervisors to see what happens. Second, the users no longer have to keep track of event name consistency between modules and between specifications. All elements of a template design have asynchronous events by default—and one need not worry about unexpected interactions. On the other hand, synchronization is not achieved by naming events consistently but rather by visually linking them. Then, it is easy to try different synchronization strategies and it is possible to use a single template instance in a number of ways without having to always rename events. This property seemed to be especially liberating since renaming events is laborious and error-prone. In our project we needed to obtain a very small supervisory solution (in order to fit in the PLC memory) and it was necessary to go through a large number of iterations where the system was simplified with different approaches. This rapid prototyping would not have been feasible if all operations had to be called manually and if event names had to be changed for every new approach. It seems that even if the use of templates is not the primary reason for applying the new methodology, users still would find it advantageous, especially in the initial stages of design.

Other observations stem from the attempts to generate working PLC code directly from the modeling software. One initial idea we had was that maybe it would be possible to store PLC code snippets together with the templates in the library. Then, the designer would not have to deal with coding at all. Unfortunately, it turns out that this would have extremely limited, if any, application. The more generic a template is (i.e., the wider application it can have), the less meaningful it is to include any code (since it loses relevance). One example of this is the "Workstation" template which was used to model both the rotating table and the arm with the grabber; these are two completely different systems, requiring completely different PLC code. Thus, it is not possible to avoid having to input custom PLC code for

every design. Another complication which we discovered was due to the design of the protocol between the custom code and the automatically generated supervisory control code. For each event, the communication happens through a shared variable. For controllable events, the variable passes information from the controller to the custom subroutine (specifies if the subroutine has to be executed), while for uncontrollable events, the variable passes information from the subroutine to the controller (specifies if the uncontrollable event has occurred). Some uncontrollable events, however, are the result of the execution of the subroutine for a controllable event. For example, the event "finish drill" is the result of the execution of "start drill". According to the protocol, the only way that the supervisory code can be informed about the occurrence of such events is through the execution of dedicated subroutines. For example, it is not possible to announce the occurrence of "finish drill" directly from within the subroutine for "start drill". The solution we devised was to use an internal variable shared between the subroutines for the controllable event and the dependent uncontrollable event. Then, the first subroutine sets this variable and the second subroutine knows that it has to announce the occurrence of the dependent event. This is exemplified next with the code for the "start drill" and "finish drill" events, which share the variable "M3.7".

```
START_DRILL
...
Network 4
LD     T102
R      M3.6, 1
S      M3.7, 1
R      Q0.4, 1
R      T102, 1
...

FINISH_DRILL
Network 1
LD     M3.7
S      <event>, 1
R      M3.7, 1
```

The practical application of theory usually is faced with many unforeseen obstacles. In the case of our example project, we discovered that the PLC unit we had available has a very limited amount of memory (4 KB). The complete system could not be implemented as envisioned originally since the PLC code could not fit in he available memory. In order to deal with this problem, we propose the use of supervisor reduction algorithms and the optimization of the generated PLC code. Due to other constraints, such as time, we did not make progress on these ideas. However, preliminary investigations showed that, using the reduction algorithms in CTCT [2], some supervisors could be reduced from thousands of states to fewer than 10 (for further information, refer to [12]).

Chapter 6

Proposed future work

The success of the development of the template design methodology and the subsequent example project does not mean that there are no open questions regarding application of supervisory theory. A few directions for future research are listed next:

- The software for template design is still under development. It needs improvement in order to allow casual use.
- The experience we gained showed that the main strength of template design is not in the availability of templates. The methodology has to be changed to suit better the needs of rapid prototyping.
- There are a number of optimizations which can be done to the the library for PLC code generation. For example, currently a single Boolean variable is used for each state of an FSA. The variable will be true if the FSA is at this state and false otherwise. Instead, a single integer variable can indicate which state of the FSA is active, if each state has a unique integer identifier. Thus, in order to keep track of 100 states, one byte can be used (one byte can handle up to 256 states), while with the current approach, each state requires one bit, thus requiring $100/8 = 12.5$ bytes.
- The current PLC code generator can output only code in the instruction list format compatible with Siemens PLCs. It is possible to design the generator using architecture similar to the one used in computer language compilers. Instead of generating in a hard-coded format, the core of the generator can produce a sort of PLC meta-language, which will be then converted into a specific PLC format by a special module. Afterwards, if one needs to select a different output format, only the converter will have to be replaced, keeping the core intact.
- The current implementation of the software supports only one-way interaction with the real system—PLC code is generated from the abstract description of the DES supervisors and it is downloaded to the PLC controller. However, it is not possible to receive any feedback from the real system when it runs. Feedback which is not real-time, such as a log of the executed events and how much time they took, may be used for analysis of the performance of the controlled system. It is much more interesting,

however, to be able to connect the software with the system controller (such as a PLC) in order to obtain real-time feedback. This could be used in many ways: from animating on the screen the execution of the system to providing high-level control from within the software, if the PLC code is equipped to delegate the control of some events to the software. In the latter case, the controller of the real system has to be able to handle delays and failures on the communication channel with the software.

Acknowledgments

We would like to thank the following people without whom this project would not have been successful: José Cury from Federal University of Santa Catarina, Brazil and Karen Rudie from Queen's University, Canada for giving me the opportunity to complete this project, in terms of time, equipment and advice; Max de Queiroz from Federal University of Santa Catarina, Brazil for helping with advice and resolving many hardware-related issues; Francisco da Silva from Federal University of Santa Catarina, Brazil for the initial implementation of the BAJ library; Steffi Klinge from Otto-von-Guericke University, Germany for the design of the DES models and channels used in the experimental application and her feedback about the process of template design; and Guilherme Lise and Luis Marques from Federal University of Santa Catarina, Brazil for creating the PLC subroutines to control the robotic testbed.

The project was supported through grants from NSERC and Queen's University, Canada and CNPq, Brazil.

Bibliography

[1] B. A. Brandin, R. Malik, and P. Malik. Incremental verification and synthesis of discrete-event systems guided by counter examples. *IEEE Transactions on Control Systems Technology*, 12(3):387–401, May 2004.

[2] CTCT software. Department of Electrical and Computer Engineering, University of Toronto, Canada. Available at http://www.control.toronto.edu/DES/.

[3] C. de Oliveira, J. E. R. Cury, and C. A. A. Kaestner. Supervisory control problem for parameterized and non regular discrete event systems. To appear in *IEEE Transactions on Automatic Control*.

[4] M. H. de Queiroz and J. E. R. Cury. Modular control of composed systems. In *Proceedings of the 2000 American Control Conference*, volume 6, pages 4051–4055, June 2000.

[5] M. H. de Queiroz and J. E. R. Cury. Synthesis and implementation of local modular supervisory control for a manufacturing cell. In *Proceedings of the 6th International Workshop on Discrete Event Systems (WODES'02)*, pages 377–382, Zaragoza, Spain, October 2002.

[6] DESUMA software. Department of Electrical Engineering and Computer Science, University of Michigan, USA. Available at http://www.eecs.umich.edu/umdes/toolboxes.html.

[7] G. Ekberg and B. H. Krogh. Programming discrete control systems using state machine templates. In *Proceedings of the 8th International Workshop on Discrete Event Systems*, pages 194–200, Ann Arbor, MI, USA, July 2006.

[8] C. M. Enright and M. Barbeau. An evaluation of the TCT tool for the synthesis of controllers of discrete event systems. In *Canadian Conference on Electrical and Computer Engineering*, volume 1, pages 241–244, Vancouver, BC, Canada, September 1993.

[9] L. Grigorov. Hierarchical control of discrete-event systems. Survey paper, School of Computing, Queen's University, Canada, 2005. Available at http://www.cs.queensu.ca/~grigorov/.

[10] L. Grigorov and K. Rudie. Problem solving in control of discrete-event systems. In *Proceedings of the European Control Conference 2007*, pages 5500–5507, Kos, Greece, July 2007.

[11] IDES software. Department of Electrical and Computer Engineering, Queen's University, Canada. Available at http://www.ece.queensu.ca/directory/faculty/Rudie.html.

[12] S. Klinge. Supervisory control of a manufacturing cell: Modeling and implementation. Pre-diploma thesis, Otto-von-Guericke University, Germany, 2007. In preparation.

[13] R. J. Leduc. *Hierarchical Interface-based Supervisory Control*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 2002.

[14] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

[15] K. Rudie. The integrated discrete-event systems tool. In *Proceedings of the 8th International Workshop on Discrete Event Systems*, pages 394–395, Ann Arbor, MI, USA, July 2006.

[16] E. A. P. Santos, J. E. R. Cury, and V. J. D. Negri. Modelagem das especificações operacionais de sistemas de manipulação e montagem automatizados. In *Símposio Brasileiro de Automação Inteligente*, pages 144–149, Bauru, São Paulo, Brazil, 2003.

[17] E. A. P. Santos, V. J. D. Negri, and J. E. R. Cury. A computational model for supporting conceptual design of automatic systems. In *Proceedings of 13th International Conference on Engineering Design*, pages 517–524, Glasgow, UK, August 2001.

[18] B. Tognazzini. *Tog on Interface*. Addison-Wesley Publishing Company, Inc., 1992.

[19] W. M. Wonham. Supervisory control theory: Models and methods. Informal talk at Queen's University, a version for the 24th International Conference on Application Theory of Petri Nets is available at http://www.control.toronto.edu/DES/publish.html, 2003.

[20] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals, and Systems*, 1:13–30, 1988.

[21] M. M. Wood. Application, implementation and integration of discrete-event systems control theory. Master's thesis, Department of Electrical and Computer Engineering, Queen's University, 2005.