Automated formal verification for flexible manufacturing systems

E. Carpanzano · L. Ferrucci · D. Mandrioli · M. Mazzolini · A. Morzenti · M. Rossi

Received: 19 October 2012 / Accepted: 15 March 2013 / Published online: 30 March 2013

E. Carpanzano
Istituto di Sistemi e Tecnologie per la Produzione Sostenibile Scuola Universitaria Professionale della Svizzera Italiana, Galleria 2, 6928 Manno, Switzerland
e-mail: emanuele.carpanzano@supsi.ch

L. Ferrucci · D. Mandrioli · A. Morzenti · M. Rossi (✉)
Politecnico di Milano, Piazza L. Da Vinci 32, 20133 Milan, Italy
e-mail: matteo.rossi@polimi.it

L. Ferrucci
e-mail: luca.ferrucci@polimi.it

D. Mandrioli
e-mail: dino.mandrioli@polimi.it

A. Morzenti
e-mail: angelo.morzenti@polimi.it

M. Mazzolini
Synesis Consortium, Kilometro Rosso Science and Technology Park, Via Stezzano 87, 24126 Bergamo, Italy
e-mail: mauro.mazzolini@synesis-consortium.eu

Introduction

Manufacturing systems are increasingly required to operate in dynamic environments characterized by quick changes of the demand, and to deliver highly customized products; this, in turn, calls for the agile and fast reconfiguration of produc-tion cells. As a consequence, the complexity of automation solutions for manufacturing systems has become consider-able. At the same time, features like interoperability, porta-bility and scalability are the key to reduce the huge costs and times needed to design and realize a new production system, or to modify an existing one. This challenging con-text requires new paradigms, based on the distribution of control onto a network of embedded components, to make the design, modification, integration and reconfiguration of resulting solution more agile (Khalgui et al. 2012). Further-more, structured approaches to control system design that support design and testing of the whole automation system must be adopted (Pranevicius 1998). Then, the guidelines, methods and tools of a comprehensive development method-ology must be defined, that allow developers to specify complex automation systems in an easy and safe way; to maintain the traceability along the different design phases; and to describe the behavior of the target system (Brusaferri et al. 2011; Basile et al. 2004). A structured methodology typically consists of the following steps:

Control system specification in which the process to be automated is described and the functional activities to be performed, as well as the purpose of the complete system are defined.

Control system architecture and functional design in which the control system is conceived and developed exploiting concepts and paradigms provided by reference models and standards.

Software implementation in which the real software solution is deployed by means of appropriate programming languages.
Verification and Validation (V&V) in which the structural correctness of the control code and the compliance of the behavior of the automation system with its requirements are verified.

In particular, the V&V phase is crucial to obtain a robust and reliable automation solution, but there is no commonly adopted effective approach for this phase. In fact, in the current industrial practice, most operating conditions of the developed system are not properly verified, and several design and implementation errors often remain unresolved until the commissioning phase due to the considerable complexity of the control logic and to the limited development time available. Nonetheless, the lack of proper identification and correction of such errors before final commissioning critically impacts on ramp-up time and costs as well as on production downtimes (Wang and Deng 1999). V&V in con-trol systems can be addressed through simulation or formal approaches. Simulation is currently the most widely known and adopted technique for V&V of industrial automation systems. The deployment and implementation of simulation frameworks is quite simple thanks to the tools available for software- and hardware-in-the-loop simulation (Zhang and Anosike 2012). The main open problem of such an approach is the definition of the test cases to achieve complete and exhaustive model analysis. Therefore, the quality of the sim-ulation results closely depends on a good definition of testing scenarios and verifying any possible behavior of the system still remains a difficult task. To overcome this limitation, formal verification approaches, which are able to exhaustively explore the execution space of a system model, have been studied and proposed for the design of manufacturing systems (Hanisch et al. 2006). In most instances, formal verification techniques are based on modeling notations that are separate from those normally used by practitioners in their design work, and the mapping from the concepts of one nota-tion to those of the other one is often difficult.

This paper presents a formal verification technique for models of manufacturing systems whose main ingredients are: (i) Stateflow diagrams as the notation for modeling the behavior of designed systems; (ii) a semantics of State-flow diagrams based on a decidable metric temporal logic;(iii) a tool capable of analyzing metric temporal logic models and of providing answers to user queries in a "push-button" manner.

In our approach, users can rely on a familiar notation (Stateflow) to describe their designs; Stateflow has been cho-sen as an example to explain the approach; other formalisms (e.g. SFC, Petri nets) could be used as well. The logic-based semantics precisely captures and resolves the intricacies (and possibly the hidden ambiguities) of the design notation, and it is used to formally check whether user-defined properties of interest are satisfied by the system model or not. In particular, thanks to the metric nature of the logic-based language underlying our approach, Stateflow models are provided with a precise, metric, notion of time; this is exploited, on the one hand, to introduce metric constraints in the models (e.g., "the plant remains in state S no longer than 3 time units"), and on the other hand to allow users to analyze properties such as "does the plant terminate the processing within 10 time units of its start?".

The paper is structured as follows: "Formal methods for the verification of automation solutions" section frames this work in the context of existing techniques, highlighting the features that separate it from them; "Tools: TRIO and ℤot" section introduces some necessary background; "Methodology overview and case study" section illustrates our approach to the verification of control designs for manufacturing sys-tems, applies it to an example system, and presents some experimental results of the verification of the example sys-tem, with emphasis on the design errors unearthed through the analysis; "Conclusions" section concludes.

Formal methods for the verification of automation solutions

Verification is the process of checking the robustness and reli-ability of the designed control solution by proving its compli-ance with a given specification (Vyatkin and Hanisch 2003). During the last few years, many research efforts focused on exploring and developing new methodologies supporting the verification process through formal approaches. Differ-ent types of formal models, as well as logics for the def-inition of the properties to be proved for the model have been investigated. Klein et al. (2002) define the model of the control system in terms of Signal Interpreted Petri Nets; these models are verified using a symbolic model check-ing tool, and are then translated into the IEC 61131-SFC language (IEC 2003). Vyatkin et al. (2003) develop a for-mal model of automation solutions based on Net Condi-tion/Event Systems (NCES); models are analyzed through SESA model checking and the properties are defined through temporal logic. Mazzolini et al. (2010) use Stateflow diagrams as the modeling notation; the properties to be proved are taken from the model coverage properties proposed by the DO 178B standard; the formal verification tool is Simulink Design Verifier. Gourcuff et al. (2008) propose a represen-tation of programs of logic controllers aiming at improving the scalability of model-checking techniques in the industrial automation domain. The benefits of this representation are shown by means of three examples using NuSMV as model checking tool. Thapa et al. (2006) translate the developed

PLC code into an intermediate language, which is then converted to Timed Automata. In this case verification is performed through the Uppaal model checker, and CTL formulae are used to define the liveness and safety properties to be checked. As described above, several formal methodologies for the verification of automation solutions have been developed. The main differences regard the types of model-checking tools exploited and the formalisms used to describe the control algorithm. Each methodology has its specific benefits and limitations, but none of the approaches mentioned above is commonly adopted in the current development practice.

The work presented in this paper addresses the verification problem by means of: (i) an intuitive semiformal notation for the description of designed controllers; we chose Stateflow diagrams because they are used by a large community of practitioners; our approach, however can be easily adapted to any state-based, possibly graphical, notation according to the preferences of the selected user community; (ii) a formal semantics of the Stateflow-based notation given in terms of a metric temporal logic; (iii) a fully-automated formal verification tool which allows users to define the system properties to be checked through formulae of the metric temporal logic mentioned above. This framework provides a high level of modeling abstraction, which allows users to formally represent the developed automation solution in a way that is at the same time adherent to the described control logics and intuitively understandable by control engineers. "Methodology overview and case study" section provides some details of the proposed approach and highlights its main benefits. In particular, it shows that the introduced logic-based technique and supporting tool allow designers to verify, on models described through a notation that is familiar to domain experts, a wide range of properties, including ones that current tools cannot tackle.

Tools: TRIO and $\mathbb{Z}$ot

TRIO (Ciapessoni et al. 1999) is a general-purpose formal specification language suitable for describing complex real-time systems, including distributed ones like Flexible Manufacturing Systems. TRIO is a first-order linear temporal logic that supports a metric on time. TRIO formulae are built out of the usual first-order connectives, operators, and quantifiers, as well as a single basic modal operator, called Dist, that relates the current time, which is left implicit in the formula, to another time instant: given a time-dependent formula F (i.e., a term representing a mapping from the time domain to truth values) and a (arithmetic) term t indicating a time distance (either positive or negative), the formula $\mathrm{Dist}(F, t)$ specifies that F holds at a time instant whose distance is exactly t time units from the current one. $\mathrm{Dist}(F, t)$ is in turn also a time-dependent formula, as its truth value can be evaluated for any time instant, so that temporal formulae can be nested as usual. While TRIO can exploit both discrete and dense sets as time domains, in this paper we assume the standard model of the nonnegative integers as discrete time domain. For convenience in the writing of specification formulae TRIO defines a number of derived temporal operators from the basic Dist through propositional composition and first-order logic quantification. Table 1 defines some of the most significant ones, including those used in this paper.

The TRIO specification of a system consists of a set of basic *items*, which are primitive elements, such as predicates, time-dependent values, and functions, representing the elementary phenomena of the system. The behavior of a system over time is formally described by a set of TRIO formulae, which state how the items are constrained and how they vary, in a purely descriptive (or declarative) fashion. The goal of the verification phase is to ensure that the system S satisfies some desired property R, that is, that $S \models R$. In the TRIO approach S and R are both expressed as logic formulae Σ and ρ, respectively; then, showing that $S \models R$ amounts to proving that $\Sigma \Rightarrow \rho$ is valid. TRIO is supported by a variety of verification techniques implemented in prototype tools. In this paper we use $\mathbb{Z}$ot (Pradella et al. 2008), a bounded satisfiability checker which supports verification of discrete-time TRIO models. $\mathbb{Z}$ot encodes satisfiability (and validity) problems for discrete-time TRIO formulae as propositional satisfiability (SAT) problems, which are then checked with off-the-shelf SAT solvers. More recently, a new encoding that exploits the features of Satisfiability Modulo Theories (SMT)(Bersani et al. 2010) solvers has been developed. Through $\mathbb{Z}$ot one can verify whether stated properties hold for the system being analyzed (or parts thereof) or not; if a property

Table 1 TRIO derived temporal operators

Operator	Definition
$\mathrm{Past}(F, t)$	$t \geq 0 \wedge \mathrm{Dist}(F, -t)$
$\mathrm{Futr}(F, t)$	$t \geq 0 \wedge \mathrm{Dist}(F, t)$
$\mathrm{Alw}(F)$	$\forall d : \mathrm{Dist}(F, d)$
$\mathrm{AlwF}(F)$	$\forall d \geq 0 : \mathrm{Futr}(F, d)$
$\mathrm{AlwP}(F)$	$\forall d \geq 0 : \mathrm{Past}(F, d)$
$\mathrm{SomF}(F)$	$\exists d \geq 0 : \mathrm{Futr}(F, d)$
$\mathrm{SomP}(F)$	$\exists d \geq 0 : \mathrm{Past}(F, d)$
$\mathrm{Lasted}(F, t)$	$\forall d \in (0, t] : \mathrm{Past}(F, d)$
$\mathrm{Lasts}(F, t)$	$\forall d \in (0, t] : \mathrm{Futr}(F, d)$
$\mathrm{WithinP}(F, t)$	$\exists d \in (0, t] : \mathrm{Past}(F, d)$
$\mathrm{WithinF}(F, t)$	$\exists d \in (0, t] : \mathrm{Futr}(F, d)$
$\mathrm{Since}(F, G)$	$\exists d \geq 0 : (\forall d' \in [0, d) : \mathrm{Past}(F, d')) \wedge \mathrm{Past}(G, d)$
$\mathrm{Until}(F, G)$	$\exists d \geq 0 : (\forall d' \in [0, d) : \mathrm{Futr}(F, d')) \wedge \mathrm{Futr}(G, d)$

does not hold, ℤot produces a counterexample that violates it. In "System properties verification and experimental results" section we will describe how we use the mechanisms implemented in ℤot to verify user-defined properties of the systems under development; this approach can be replicated in any tool environment, such as the symbolic model checker NuSMV (Cimatti et al. 2002), that is capable of verifying linear-time temporal logic specifications.

Methodology overview and case study

We illustrate our approach through a robotic cell of a flexible manufacturing system (FMS). The cell includes a robot arm that loads and unloads pallets of two different types on two distinct machines. The cell, as shown in Fig. 1, is served by a conveyor belt (*Conveyor_in*), which provides both pallets of type A, to be processed by *Machine 1*, and of type B, to be processed by *Machine 2*. The finished artifacts are released by the cell through the conveyor out belt (*Conveyor_out*).

The control solution for the robotic cell is deployed according to the IEC 61499 standard (Lewis 2001; IEC 2005), which defines function blocks for industrial process measurement and distributed control systems. The standard is based on a fundamental type of module, the *Function Block* (FB), which represents a software functional unit, associated with a hardware resource of the control system. Standard IEC 61499 plays a crucial role in supporting the development of distributed control solutions for flexible manufacturing systems (Vyatkin 2011) where each control module encapsulates control logic defined by an *Execution Control Chart* (ECC), consisting of states, transitions and actions,

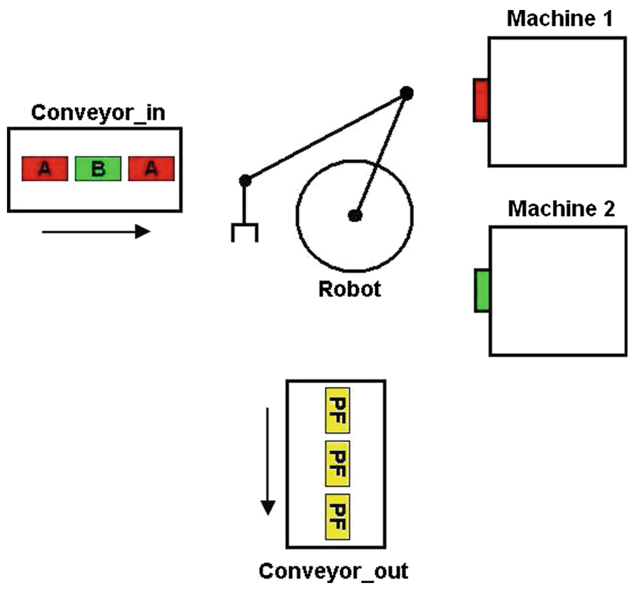


Fig. 1 Robotic cell

which invokes the execution of algorithms in response to input events. An application can be distributed over several control system devices. A device uses the causal relationship specified by the application to determine the appropriate responses to events. Furthermore, in the IEC 61499 standard a resource is a software (and possibly hardware) component of a device, which has independent control of its operations. Each FB instance is associated with one single resource. The control solution for the robotic cell is implemented using the ISaGRAF6 environment (ISaGRAF 2012), which completely supports the development of control applications with the IEC 61499 standard. The developed IEC 61499 control solution is shown in Fig. 2 and the ECC encapsulated within the robot controller module is depicted in Fig. 3

The adoption of the IEC 61499 standard as reference model for the control system of the robotic cell fosters the definition of reusable control modules and the re-configurability of the solution, since the principles of modularity, encapsulation and standardization of interfaces are effectively supported. The control logic of each component of the FMS is translated into Stateflow diagrams, from the developed IEC 61499-compliant control solution, for its formal verification. To guarantee that the properties and the features of the IEC 61499 control solution described above are maintained in the Stateflow diagrams, the rules defined in Ballarino and Carpanzano (2002) to translate an IEC 61499 model into a corresponding Stateflow are exploited. More precisely, the Simulink Stateflow description of an IEC 61499 model is obtained by describing each FB through a Simulink block where:

- Input and output data are represented as input and output signals of Boolean and Double types in the corresponding Simulink block.
- The ECC and related algorithms are represented by means of Stateflow diagrams, which can call Matlab functions.
- Input and output events are represented as rising or falling edges of input and output signals of Boolean type in the corresponding Simulink block.
- Internal data is represented by means of local Simulink variables.

Finally, the Stateflow model is obtained by connecting the Simulink blocks according to the structure of the original IEC 61499 model.

In FMSs non-deterministic choices within a component must be avoided. In our example, to deal with multiple requests from different components we statically assign priorities to the operations performed by the robot: the unloading of workpieces from the machines has higher priority than their loading; also, unloading pieces from *Machine 1* has precedence over unloading from *Machine 2*. Finally, at

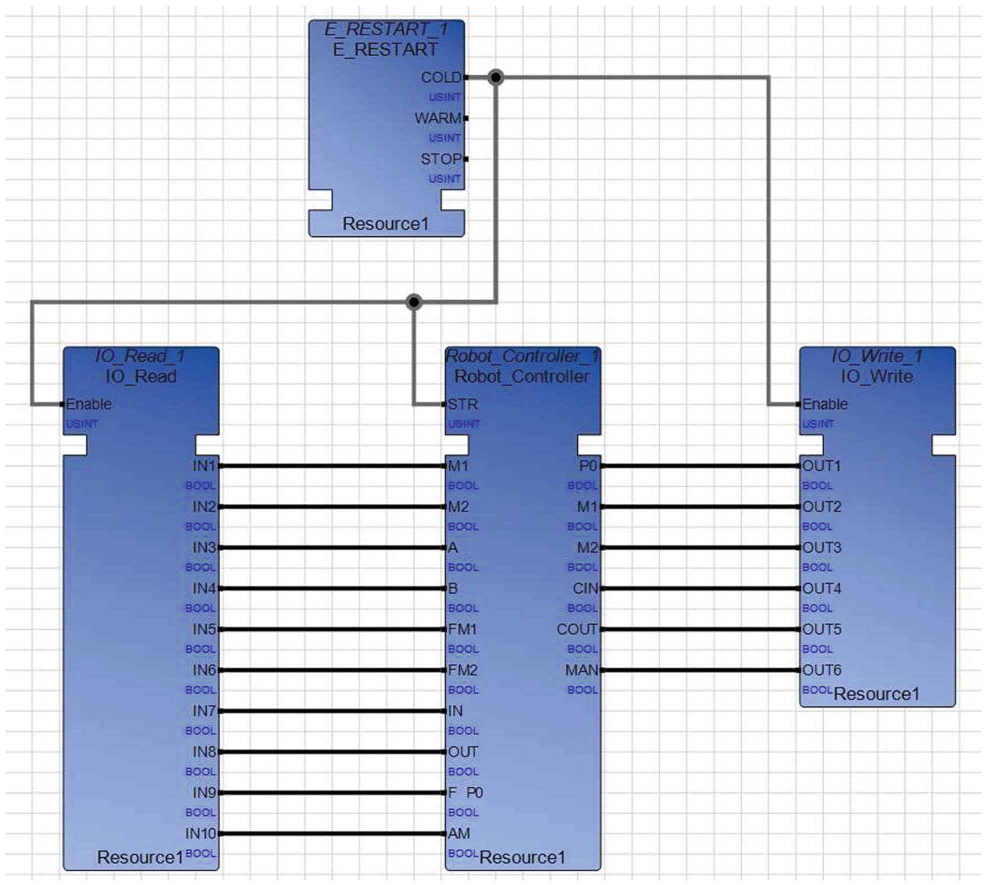


Fig. 2 Developed IEC 61499 control solution

any time the robot arm can switch from automatic to manual mode, where an operator can send commands directly to the robot when the need arises to perform operations outside the production cycle. The system switches back to automatic mode through a suitable command.

The Stateflow diagrams of Figs. 4 and 5 provide a model of the behavior of components *Robot* and *Machine 1*, respectively. These diagrams are composed through the Simulink graph of Fig. 6 to define the model of the robotic cell.

The Stateflow notation is a variation of Statecharts (Harel 1987). A Stateflow diagram (Mathworks 2011) is composed of:

1. A finite set of typed variables $D = D_I \cup D_O \cup D_L$. D is partitioned into input variables D_I, output variables D_O, and local variables D_L. D_I and D_O include Boolean variables used to represent input and output events: a variable v_i (resp. v_o) modeling an input (resp. output) event is set to $true$ when the event is received from (resp. notified to) the environment.
2. A finite set of states S. A state can be associated with three kinds of *actions*: *entry*, *exit* and *during* actions; they are executed, respectively, when the state is entered, exited, or during the permanence of the system in the state. An *action* is the assignment of the value of an expression over constants and variables of D to an output or local variable.
3. A finite set of transitions, T, that may include guards (i.e., constraints) on the variables of D and actions.

Actions over both states and transitions allow one to write Stateflow diagrams in a more concise way, since it is possible to build a semantically equivalent Stateflow diagram with actions over transitions only. For example, an *entry* action of a state is equivalent to an action on any transition entering the state; a during action of a state s with a transition from s to itself is equivalent to a *during* action on the transition.

Simulink diagrams (Mathworks 2011) are used to compose modules evolving in parallel into new components. Components can be *basic* or *composed*. A basic component has a public interface, and its behavior is described through a Stateflow diagram. The public interface comprises the set of variables $D_{Int} = D_I \cup D_O$. Composed components are built from basic ones in a hierarchic manner. At the lowest level of the hierarchy, a composed component is described by a Simulink graph with two or more basic components. The interface of a composed component is the union of the

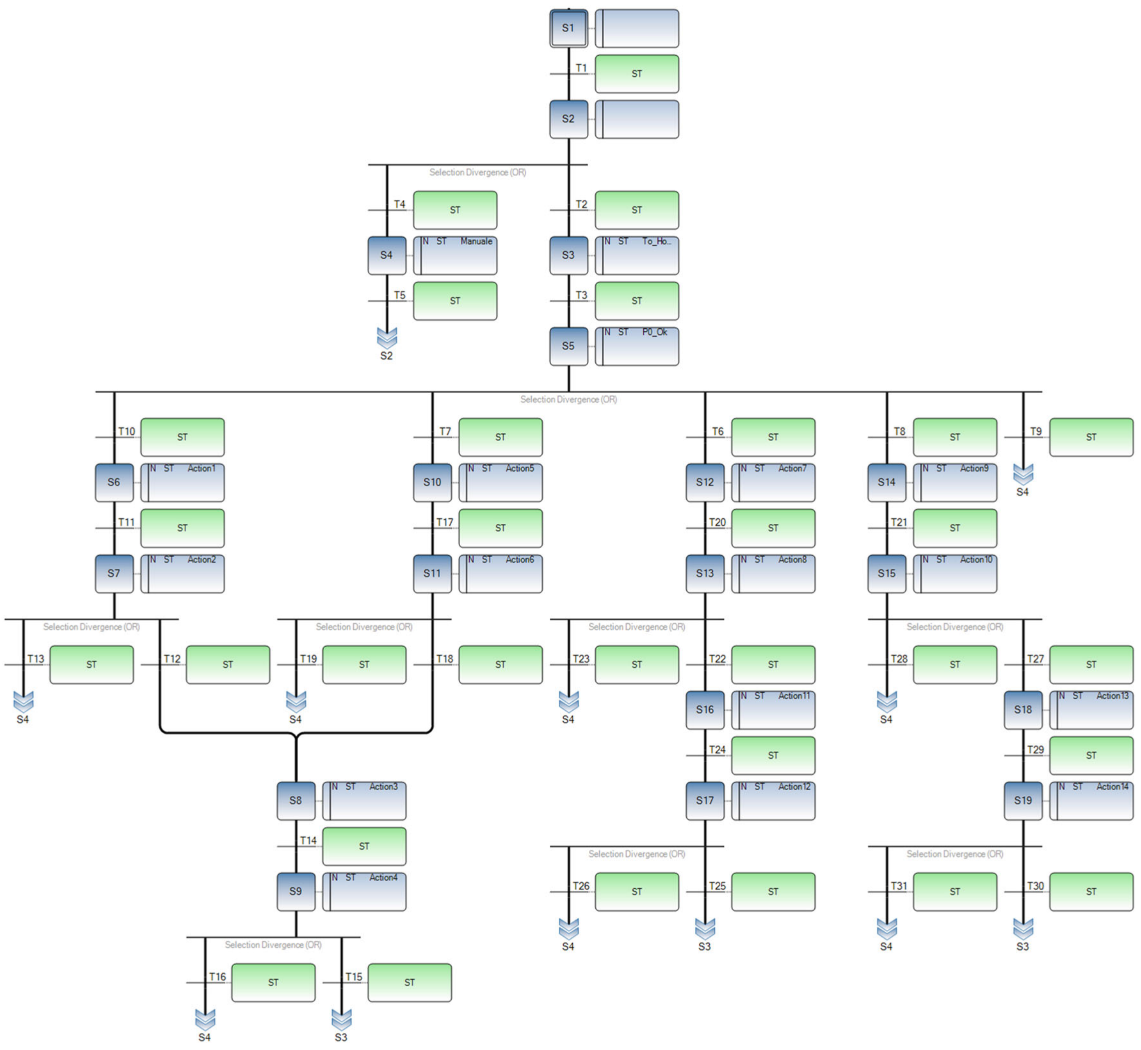


Fig. 3 ECC of the robot controller module

input and output variables of its parts, while its behavior is described by the Stateflow diagrams of its modules, whose communications are represented graphically through *links*. Each link corresponds to a flow of messages (signals or data) sent from a component to another. The communication is realized through the assignment of the value of an output variable of the sending component to an input variable of the receiving one. Simulink diagrams can in turn be composed to obtain higher-level components. The detailed features of communication are explained in the coming "Semantics" section.

Semantics

Our semantics of Stateflow diagrams is based on the STATE-MATE semantics of Statecharts (Harel and Naamad 1996). It includes a composition operator for building hierarchical, modular models from simpler ones.

The semantics of a Stateflow diagram is a set of **runs**, representing the reaction of the actual system to a sequence of input events. A run is a sequence of **configurations** $\{c_i\}_{i \geq 0}$ such that, for each $i > 0$, c_i is obtained from c_{i-1} by executing a step. A configuration c_i is a pair $\langle s_i, \mu_i \rangle$ where $s_i \in S$ is the currently active state and μ_i is a *valuation* of the variables of D, i.e. a mapping $\mu_i : D \rightarrow dom(D)$. Expressions on the variables of D are evaluated in the active configuration. A transition $s \overset{g/a}{\rightarrow} s'$ from state s to s' with guard g and action a is **enabled** in a configuration $c = \langle s, \mu \rangle$ only if g is true for μ; the execution of the transition from c produces a new configuration c' that is obtained by applying action a to μ. A transition *must* be executed as soon as it is enabled, hence there cannot be more than one transition enabled in the same configuration, or the model becomes inconsistent (this can be resolved by prioritizing transitions, as mentioned above).

The semantics of Statecharts (and also of Stateflow) has proven difficult to pin down precisely, and different solutions

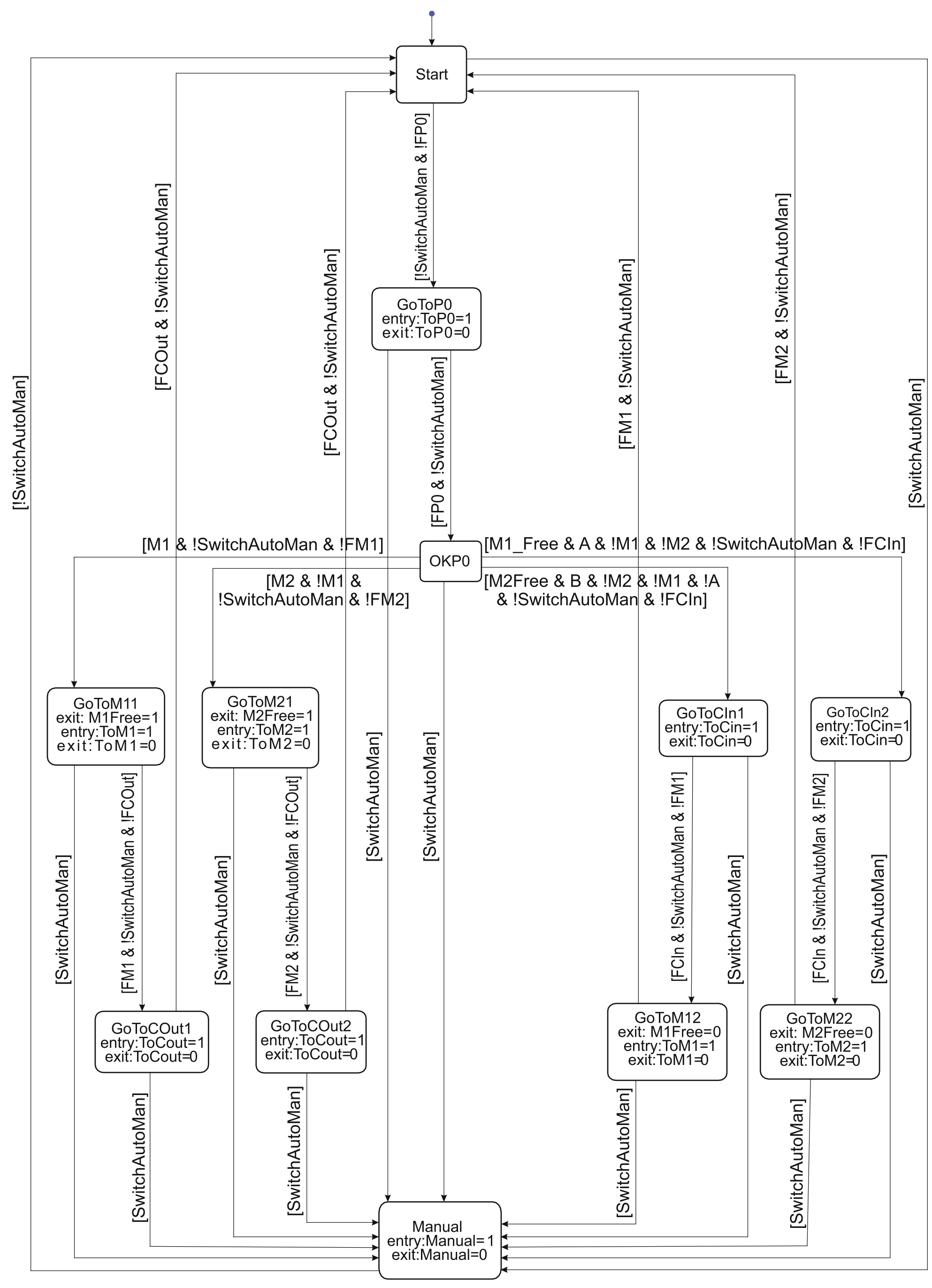


Fig. 4 Stateflow diagram of the controller of the robotic cell of Fig. 1

have been proposed (e.g., Levi 2000; Alur and Henzinger 1999). Our model is of the so-called **run-to-completion** kind. The system reacts to the input events by performing a sequence of actions called **macro-steps**. In every macro-step, a maximal set of enabled transitions is selected and executed based on the events generated in the previous macro-step. We call **micro-step** the execution of a transition within a macro-step. Conventionally, micro-steps take zero time to execute; when no transition is enabled the system reaches a **stable** configuration, a new input event is received from the environment, and time advances to the next macro-step. As in the STATEMATE semantics of Statecharts, components sense input events and data only at the beginning of macro-steps. They communicate output events and data to the environment only at the end of macro-steps. In summary, the semantics of a macro-step is as follows:

1. When a macro-step begins, input data and events are assigned to the corresponding variables of set D_I. Suppose for example that the current configuration c_i of the Robot Stateflow of Fig. 4 is $c_i = \langle OkP0, \{M1Free = 1, M2Free = 0, FM2 = 0, M1 = 0, \ldots\}\rangle$, and that the input event $In2$ is active, meaning that there is a completed workpiece on *Machine 2*. Then the input variables are updated to false except $M2$, producing a new configuration c_{i+1} with the current time and state unchanged.
2. As long as there are enabled transitions, micro-steps are executed in zero time. For example the transition enabled in configuration c_{i+1} is the one with guard $[M2\ \&\ !M1\ \&\ !SwitchAutoMan\ \&\ !FM2]$, so the transition is made and the system executes action $entry$: $ToM2 = 1$ of state $GoToM21$, leading to the new configuration $c_{i+2} = \langle GoToM21, \{ToM2 = 1, \ldots\}\rangle$. As before, time does not advance.
3. When there are no more enabled transitions to execute, a stable configuration is reached. At this point the macro-step is completed, time advances one unit, and output events and data produced during the macro-step are communicated to the environment. In our example, no transition is enabled in configuration c_{i+2}, so time advances, the values of the variables and the current state do not change and the new event $ToM2$ is produced according to the Simulink graph of Fig. 6.

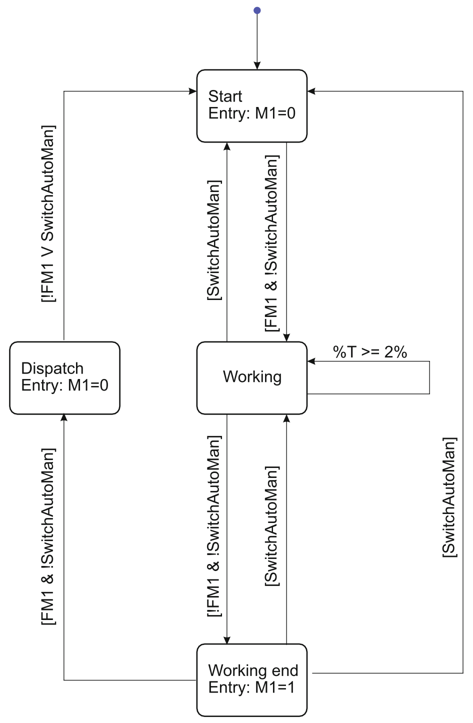


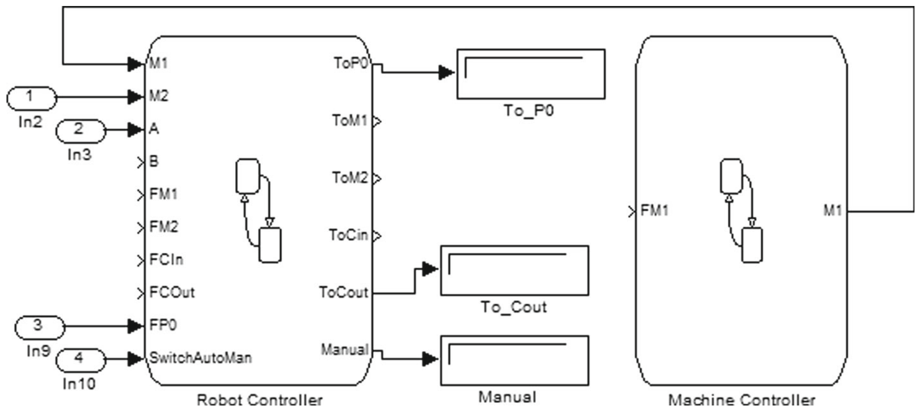
Fig. 5 Stateflow diagram of component *Machine 1* of Fig. 1

A run identifies a sequence of time instants $\{T_i\}_{i\in\mathbb{N}}$, one for each macro-step, hence the time domain is discrete. This is consistent with the underlying physical model, as the PLCs on which FMS control solutions are built are governed by discrete clocks, i.e. each macro-step corresponds to a **clock cycle** of the modeled PLC.

The run-to-completion semantics is based on the assumption that the reaction of the system to the input events is instantaneous. The following conditions, which are reasonable in practice for a large class of systems, including FMSs, capture the required assumptions: (i) the environment can be described as a discrete process, namely as an infinite sequence of inputs $\{I_i\}_{i\in\mathbb{N}}$ occurring at successive instants of time; (ii) the system is infinitely faster than the environment, so its reaction to inputs I_i is completed before inputs I_{i+1} are produced.

The definition of step in Statecharts (and similar notations) has long been debated in the literature, since the assumption of instantaneous reaction to inputs produces some paradoxes and unexpected situations (Levi 2000), and in particular it allows for the presence of **Zeno runs**. A run has Zeno behav-ior if infinitely many actions are executed in a finite amount of time. In Stateflow diagrams, this corresponds to the situation in which infinitely many micro-steps are executed during a single macro-step, which in turn occurs when the run enters a loop of micro-steps that is never exited, thus never triggering

Fig. 6 Simulink diagram of the robotic cell



the advancement of time. Zeno runs must be avoided in models because they represent unfeasible behaviors. "System properties verification and experimental results" section shows how Zeno runs can be detected using our formal semantics of Stateflow.

The last part of the semantics concerns the composition of two or more modules according to the Simulink graph. Given two Stateflow diagrams $G_1 = \langle D_1, S_1, T_1\rangle$ and $G_2 = \langle D_2, S_2, T_2\rangle$, we introduce a compositional binary operator $\|$ whose result is a new component $G = \langle D, S, T\rangle$ where $D = D_1 \cup D_2$, $S = S_1 \times S_2$. If we denote the set of all possible configurations of G_1 and G_2 with C_1 and C_2, respectively, and the evaluation of their input variables with D_{I_1} and D_{I_2}, the transition relation of the composed component G is a function $\rightarrow: C_1 \times C_2 \times D_{I_1} \times D_{I_2} \longmapsto C_1 \times C_2$ defined by the following rules:

1. Suppose that G is in configuration $c = \langle c_1, c_2\rangle$, with $c_1 = \langle s_1, \mu_1\rangle \in C_1$ and $c_2 = \langle s_2, \mu_2\rangle \in C_2$. If there are two transitions $s_1 \overset{g_1/a_1}{\rightarrow} s_1'$ and $s_2 \overset{g_2/a_2}{\rightarrow} s_2'$ of, respectively, G_1 and G_2 that are enabled in c_1 and c_2, then the system moves to the new configuration $c' = \langle c_1', c_2'\rangle$ where $c_1' = \langle s_1', \mu_1'\rangle$ and $c_2' = \langle s_2', \mu_2'\rangle$. μ_1' and μ_2' are the new evaluations of the variables according to the execution of actions a_1 and a_2. In this case both components execute their transition in a real parallel manner and time does not advance.
2. Suppose that G is in configuration $c = \langle c_1, c_2\rangle$, but only the transition $s_1 \overset{g_1/a_1}{\rightarrow} s_1'$ is enabled. In this case the system moves to the new configuration $c' = \langle c_1', c_2\rangle$ (with $c_1' = \langle s_1', \mu_1'\rangle$), i.e., the second component does not change its local configuration, thus executing a *stutter* ε transition, in which the current state is repeated. Time does not advance, since the first component executes a non-stutter transition.
3. Suppose finally that G is in a configuration $c = \langle c_1, c_2\rangle$, but none of the transitions with source states s_1 and s_2 are enabled in G_1 and G_2. In this case, both components have reached a stable configuration. The new configuration c' is the same as c, except that time advances one unit. All the output events and data are produced and sent as input events and data to the corresponding receiving components according to the Simulink graph links.

From the above description, it follows that the clock of the composed module G synchronizes the clocks of its components, and a component can reach a stable configuration before the others. Then, each component that is in a stable configuration must perform stutter transitions until all other components also reach a stable configuration. This mechanism is exemplified in Fig. 7, which shows the fragments of the runs of two modules A and B that are composed to realize a third component C. The figure shows the first macro-step of the two runs: for component A the macro-step begins in state S_0 and ends in state S_3; similarly for component B. The x axis shows the number of micro-steps executed from the beginning of the macro-step: component B reaches a stable configuration in state S_9 in fewer micro-steps than A, so a stutter ε transition is introduced to synchronize the clocks of the two components. After the fourth micro-step, both components have reached a stable configuration and the clock of component C advances to the next time unit.

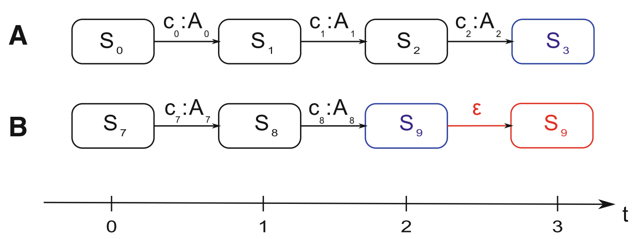


Fig. 7 Example of stutter transitions

The composition operator $\|$ is such that, when two modules are composed, their input and output variables become input and output variables of the composition, i.e., they do not become local to the composed module, hence they cannot be hidden. In addition, an input variable cannot be linked to more than one output variable, to guarantee the uniqueness of the value assigned to the input variable (on the other hand, an output variable can be linked to more than one input variable, as in a "multicast" communication). It is easy to prove that, thanks to the restrictions above, the parallel composition operator is associative, so one can build a hierarchy of components. Each component is a black box module that an engineer can design and verify separately through $\mathbb{Z}$ot by a compositional verification approach. These modules can then can be composed into Simulink graphs using only their public interfaces.

Temporal logic encoding

The semantics of Stateflow diagrams is formalized in temporal logic. We use the qualitative temporal operators of LTL (Furia et al. 2012) to describe the sequence of micro-steps, and the metric operators specific of TRIO to express quantitative properties of macro-steps. Ultimately, the semantics is expressed in the input language of the $\mathbb{Z}$ot tool, thus allowing users to perform automatic verification of Stateflow models. For each variable $V \in D$ of the Stateflow model we intro-duce a corresponding $\mathbb{Z}$ot variable with domain $dom(V)$. When variables have the same domain, we group them in $\mathbb{Z}$ot *arrays* (i.e., finite sequences of variables that are accessed through an index). In the case of the controller of the robotic cell of Fig. 4 we introduce three arrays, $InputCRO$ (of 10 elements), $OutputCRO$ (of 6 elements) and $LocalCRO$

(of 2 elements), corresponding, respectively, to the sets D_I, D_O and D_L of the Stateflow model. We also introduce a $\mathbb{Z}$ot variable V_S representing the current state of the Stateflow diagram, whose domain $dom(V_S)$ corresponds to the set S of states. In the case of the diagram of Fig. 4, $StateCRO$ is a variable with domain $[0, \ldots, 11]$, where each value corresponds to a different state and 0 is the initial state. We use temporal logic formulae to define constraints defining valid sequences of micro-steps.

For each Stateflow transition $T_i : s_i \overset{g_i/a_i}{\rightarrow} t_i$ originating from state s_i and targeting state t_i (with $s_i \neq t_i$) with guard g_i, we introduce the following formula:

$$\Gamma_i \wedge (S = s_i)) \Rightarrow \bigcirc(S = t_i) \wedge A_i \wedge A_{ex_{s_i}} \wedge A_{en_{t_i}} \quad (1)$$

where $\bigcirc$ is the usual LTL **next state** operator (i.e., $\bigcirc F$ holds in the current state iff F holds in the next state), Γ_i is a Boolean formula encoding guard g_i, and A_i, $A_{ex_{s_i}}$ and $A_{en_{t_i}}$ are temporal logic formulae encoding, respectively, the transition action a_i, and the *entry* and *exit* actions of states s_i and t_i. The formula asserts that if the current state is s_i and the transition condition Γ_i holds in the current configuration, then in the next micro-step the active state must be t_i and the $entry$ actions of state t_i and the $exit$ actions of state s_i are executed. In addition, if no transition is enabled, the configuration does not change, which is captured by the following formula:

$$\left(\bigwedge_{i=1}^{N} \neg(\Gamma_i \wedge (S = s_i)) \right) \Rightarrow NOCHANGE \quad (2)$$

where N is the number of transitions of the diagram, and subformula $NOCHANGE$, which we do not detail here, asserts that in the next micro-step the current state and the values of all output and local variables do not change. When no transition is enabled, the model reaches a **stable state**. The complete definition of the behavior of the transitions of the Stateflow diagram is given by $\left(\bigwedge_{i=1}^{N}(1)_i\right) \wedge (2)$.

The time advancement of our run-to-completion semantics is modeled by a predicate called $tick$, which is added to the encoding of each Stateflow diagram. Predicate $tick$ holds in each micro-step following one in which the diagram has reached a stable state. When predicate $tick$ is true, time advances to the next clock cycle. The behavior of predicate $tick$ is captured by the following formula:

$$\left(\bigwedge_{i=1}^{N} \neg(\Gamma_i \wedge (S = s_i)) \right) \Leftrightarrow \bigcirc tick \quad (3)$$

To foster information hiding, instead of relying on a single global predicate modeling time advancement, in our encoding each module has its local $tick$ predicate. As explained below, additional predicates and formulae are introduced in the composed system model for the synchronization of the local ticks.

We introduce a formula asserting that when predicate $tick$ is false the values of the input variables D_I must be the same as in the preceding micro-step:

$$\Box(\neg tick \Rightarrow \left(\bigwedge_{v \in D_I} (\forall x \overleftarrow{\bigcirc}(v = x) \Rightarrow (v = x)) \right) \quad (4)$$

where the $\overleftarrow{\bigcirc}$ and $\Box$ are, respectively, the **yesterday** and **globally** LTL operators: $\overleftarrow{\bigcirc} F$ holds if formula F held the previous state, while $\Box F$ holds if F is true in the current and in all future states. The conjunction of formulae $\bigwedge_{i=1}^{N}(1)_i$, (2–4), is a formula MOD that characterizes all the runs of the Stateflow diagram, i.e. it encodes the module's behavior.

Next we encode the semantics of module composition described through Simulink graphs. We use a modular approach to hide the details of time advancement of modules to other components. A special integrator module M is added that can access the data of the public interfaces of each component and includes a number of axioms to ensure the correct behavior of the composed module as defined in the Simulink graph.

As described in "Semantics" section, the local clock of a component that is part of a bigger system advances only when the latter has reached a stable configuration. However the local $tick$ predicate of the module does not convey the information on when such event occurs, since it is not directly related to the state of the other components. To avoid the use of a global $tick$ predicate shared between components, which would break compositionality, we add two new local predicates to the interfaces of each component module, $stable$ and $tick^{\text{ext}}$. $stable$ is true when the component reaches a locally stable configuration and it replaces predicate $tick$ in formula (3), thus giving the new formula:

$$\left(\bigwedge_{i=1}^{N} \neg(\Gamma_i \wedge (S = s_i)) \right) \Leftrightarrow \bigcirc stable \quad (5)$$

We use predicate $tick^{\text{ext}}$ to convey to single modules the information about the overall system state. This predicate is set to true by the integrator module M when all its components reach a stable configuration. In our compositional semantics the local $tick$ of each component is true iff both $stable$ and $tick^{\text{ext}}$ are true, which is captured by the following formula:

$$\Box(tick \Leftrightarrow stable \wedge tick^{\text{ext}}) \quad (6)$$

Module M has also its own local clock predicates $tick_M$, $stable_M$ and $tick_M^{\text{ext}}$, so the semantics is compositional in that M itself can be part of bigger modules, but this does not affect its formulae. If M is the composition of n_M modules, to obtain the synchronization of the clocks of the component modules, the following three conditions must hold:

- Predicate $stable_M$ is true iff all the predicates $stable_i$ of the component modules are true. This condition implies that the local clock of the composed module becomes true iff the overall system is in a stable configuration.
- Each predicate $tick_i^{\text{ext}}$ of each component module is equal to $tick_M$. This condition implies that all the local clocks of the component modules become false if the local clock of the composed module M is false.
- Predicate $tick_M$ is true iff predicates $stable_M$ and $tick_M^{\text{ext}}$ hold, as for the local components.

The three conditions above are formalized through the following formulae of module M:

$$\Box\left(stable_M \Leftrightarrow \left(\bigwedge_{i=1}^{N} stable_i\right)\right) \tag{7}$$

$$\Box\left(\bigwedge_{i=1}^{N} \left(tick_M \Leftrightarrow tick_i^{\text{ext}}\right)\right) \tag{8}$$

$$\Box(tick_M \Leftrightarrow stable_M \wedge tick_M^{\text{ext}}) \tag{9}$$

Finally, we formalize the semantics of the relations between components represented as links in the Simulink graph, as described in the "Methodology overview and case study" section. A link between an output variable of a component i and an input variable of a component j means that the corresponding data or events produced by i are sent to j. This corresponds to synchronizing the value of the input variables of j to the value of the output variables of i only when the overall system has reached a stable configuration, i.e. when the predicate $tick_M$ is true. This is captured by the following formula

$$\Box(tick_M \Rightarrow (vout_{i_1} = vin_{j_1})) \tag{10}$$

where $vout_{i_1}$ is an output variable of component i linked to the vin_{j_1} input variable of the component j. M contains an instance of formula (10) for each link of the Simulink graph.

In our compositional semantics, the formula SYS encoding the behavior of the composed system is given by the conjunction of the local formulae $\bigwedge_{i=1}^{N}(1)_i$, (4–5) for each component, plus the formulae (6–10) introduced in module M.

We remark that, since in the run-to-completion semantics time advances only when macro-steps are performed, to express metric properties of systems using the TRIO language we need to redefine TRIO operators to reflect this notion of time advancement. We redefine the TRIO Dist operator, where $\text{Dist}(F, K)$ holds in each instant t such that formula F holds at time $t + K$. The following formula defines the meaning of $\text{Dist}(F, 1)$, to take into account that time advances only at the occurrence of $tick$, starting from the operators predicating on micro-steps:

$$\text{Dist}(F,1) = \bigcirc(\neg tick \mathsf{U}(tick \wedge \bigcirc(\neg tick \mathsf{U}(\bigcirc tick \wedge F)))) \tag{11}$$

where U is the usual binary LTL operator **until**, and $A \mathsf{U} B$ holds in those states such that there is a future state in which B holds, and A holds in all states up to that one (excluded). Formula (11) asserts that $\text{Dist}(F, 1)$ holds if, at the end of the macro-step following the current one, F holds. The end of the current macro-step occurs the micro-step right before the next state in which $tick$ holds (i.e., the future micro-step in which $\bigcirc tick$ is true and such that $tick$ is false in all states in-between). $\text{Dist}(F, K)$ is therefore defined as $\text{Dist}(\ldots \text{Dist}(\text{Dist}(F, 1), 1 \ldots, 1))$ (k times).

We also redefine the Until TRIO operator, to predicate only on macro-steps. $\text{Until}(A, B)$ is defined by the following formula:

$$\text{Until}(A, B) = \bigcirc((\bigcirc tick \Rightarrow A)\mathsf{U}(\bigcirc tick \wedge B)) \tag{12}$$

Formula (12) asserts that $\text{Until}(A, B)$ holds if there is a future state in which the clock ticks, B holds at the end of that macro-step, and A holds at the end of all macro-steps in between. Notice that we evaluate formulae A and B only in the last micro-step of a macro-step, when all system variables have certainly been updated.

Sf2Trio: a tool to encode Stateflow/Simulink models into Trio

Sf2Trio is a prototype tool that supports designers in the construction of Stateflow/Simulink models, avoiding the burden to write manually the necessary TRIO formulae. It constitutes a first step towards the development of a completely automatic tool to translate a system specified by a Stateflow/Simulink diagram into a set of TRIO formulae that characterize its behavior, using the encoding given in "Temporal logic encoding" section to enforce the semantics of "Semantics" section.

Sf2Trio has been developed as a plug-in for the $\mathbb{Z}$ot bounded model checker (Pradella et al. 2008). The current prototype version of the tool provides a set of commands to specify a Stateflow diagram for a basic component and to define the composition and interactions of several basic components into a composed one.

We first illustrate the commands to define the Stateflow diagram of a basic component. Each command is specified by its syntax, with the name of the command in bold and parameters in italic (parameters with a colon, as in *:parameter-name*, are optional).

1. (**make − module** $ModName$)
   A *module* is the entity that, in Sf2Trio, represents a (basic or composed) component. Parameter *ModName* is the module name.
2. (**def − variable** $VName\ Type\ DomType\ \ :range$ $(minvalue\ \ maxvalue)$)
   Command **def-variable** defines a single Stateflow variable, with the following parameters:
   - *VName* is the name of the variable.
   - *Type* is one of three predefined values (***Input***, ***Output*** and ***Intern***) qualifying the variable as input, output or internal.
   - *DomType* denotes the type of the variable; in the present version of the tool only integer (***Int***) and Boolean (***Bool***) variables may be declared; integer variables range over finite domains.
   - *:range* specifies the range of an integer variable.
3. (**def − state** $SName\ IsInit\ :entering\ Acts\ :during$ $Acts\ \ :exiting\ \ Acts$)
   Command **def-state** defines a single state of the Stateflow diagram. Its parameters are the following:
   - *SName* is the state identifier.
   - *IsInit* is a Boolean value indicating the unique initial state for the Stateflow diagram.
   - *:entering*, *:during* and *:exiting* each specify a list of **actions** *Acts* that are executed when the state is respectively entered, exited, or throughout the permanence in the state. The syntax of a single action takes the form (*var expression*) where *expression* denotes the value assigned to *var*. Arithmetic operators include the usual $+$, $*$, $-$ and $/$; constant expressions must be integers.
4. (**def − transition** $SourceSName\ DestSName\ \ :condition\ Expr\ :action\ Act$)
   Command **def-transition** defines a single transition of the Stateflow diagram, with the following parameters:
   - *SourceSName* and *SourceDestSName* are, respectively, the label names of the source and destination states of the transition.
   - *:condition* specifies the transition condition as a Boolean expression *Expr*. If absent, the condition is always true.
   - *:action* specifies a transition action, which is executed when the transition is enabled.

The following Sf2Trio commands are used to declare a composed component.

1. (**make − composed − module** $ModName\ Composed$ $Modules$)
   Command **make-composed-module** specifies the set of components of the specified composed module.
   - *ModName* is the name of the composed module.
   - *ComposedModules* is a list of pairs ($ModName$ $Path$), where the first parameter is the name of a module to compose and the second is the file that contains its specification.
2. (**def − connection** $SourceVName\ \ DestVName$)
   Command **def-connection** defines an equivalence between a pair of variables (*SourceVName*, *DestVName*); this corresponds to a *link* of the Simulink graph that describes the composed module. The connected variables must be of the same type, but of different modules.

The following commands, which can be declared in any module, are used to build the formal model of the module and to verify user-defined properties thereof.

1. (**make − model**)
   Command **make-model** builds and returns the temporal logic formula that represents the module, which can then be used to perform V&V.
2. (**def − axiom** $Formula$)
   Command **def-axiom** defines a TRIO temporal logic formula that specifies a temporal property of the model, e.g. a constraint that a machine must remain in a busy state for a fixed number of time instants, which corresponds to its working time.
   - *Formula* is the TRIO formula, written in the input language of $\mathbb{Z}$ot, that formalizes the desired property. States and variables can be referred to in the formula using the corresponding predicates automatically introduced by the Sf2Trio tool, which have the form $moduleName\_elemName$. For example, formula $\bigcirc Mod\_Start$ defines that, in the next time instant, module Mod must be in state $Start$.
3. (**zot** $BoundLength\ \ Property$)
   Command **zot** is the interface to the $\mathbb{Z}$ot bounded model checker. It performs the verification of a user-defined property for the specified module.
   - *BoundLength* is the maximum length of runs analyzed by $\mathbb{Z}$ot. It is the length of execution traces analyzed by the solver.
   - *Property* is the user-defined property, written in TRIO, that is to be analyzed by $\mathbb{Z}$ot.

Figure 8 is the Sf2Trio model of the Stateflow diagram of the *Machine 1* of Fig. 5.

Fig. 8 Model of Machine 1 in the input syntax of the Sf2Trio tool

```
(make-module 'Machine1)

;;; Input Variables Definitions
(def-variable 'FM1 '*Input* '*Bool*)
(def-variable 'SwitchAutoMan '*Input* '*Bool*)

;;; Output Variables Definitions
(def-variable 'M1 '*Output* '*Bool*)

;;; State Definitions
(def-state 'Start t :entering ((M1 0)))
(def-state 'Working nil)
(def-state 'WorkingEnd nil :entering ((M1 1)))
(def-state 'Dispatch nil :entering ((M1 0)))

;;; Transition Definitions
(def-transition 'Start 'Working :condition (and FM1 (not SwitchAutoMan)))
(def-transition 'Working 'Working )
(def-transition 'Working 'Start :condition SwitchAutoMan)
(def-transition 'Working 'WorkingEnd :condition (and (not FM1) (not SwitchAutoMan)))
(def-transition 'WorkingEnd 'Working :condition SwitchAutoMan)
(def-transition 'WorkingEnd 'Start :condition SwitchAutoMan)
(def-transition 'WorkingEnd 'Dispatch :condition (and (FM1 (not SwitchAutoMan))))
(def-transition 'Dispatch 'Start :condition (or (not FM1) SwitchAutoMan))
```

System properties verification and experimental results

We now illustrate how the encoding presented in "Temporal logic encoding" section can be exploited to check some relevant properties of the robotic cell of Fig. 1. The formulae analyzed in this section capture but a portion of the kinds of properties that can be checked through our approach; they show how the technique presented in this paper can be applied to study a wide range of features of modeled systems.

We first check that the modeled system does not have Zeno runs, which would make it unfeasible. The system shows a Zeno behavior if, from a certain point on, time does not advance, i.e., predicate $tick$ does not hold. The *presence* of Zeno runs is formalized by the formula:

$$\Diamond(\Box(\neg tick_M)) \tag{13}$$

where $tick_M$ is the global tick predicate of the robotic cell, and $\Diamond$ is the **eventually** LTL operator. $\Diamond F$ holds in a state if there is a future state in which formula F holds. Formula (13) holds if, from a certain micro-step on, the clock does not tick any more. We checked through the $\mathbb{Z}$ot tool that formula $SYS \wedge$ (13) is *unsatisfiable*, which means that there are no runs of the system that also show property (13), hence the model does not exhibit Zeno runs.

Since we are now guaranteed that time advances in the modeled system, we can use the TRIO temporal operators to predicate on actual time instants (i.e., on macro-steps), and state metric properties such as "operation OP terminates within K time units", etc.

Next, we check for the existence of deadlocks in the system model. A model is *deadlock-free* if it cannot reach a configuration after which its state does not progress anymore. The usual definition of deadlock requires that the model *never* leaves its state s; in this case, however, we only consider what happens at the end of macro-steps, and we ignore the intermediate micro-steps. In other words, the deadlock is defined over macro-steps only, considering internal micro-steps states as transient states, non-observable outside the module. Different analyses would have been possible with a simple tweak of the formulae checked. The presence of deadlock does not depend on the value of the input data since we have a closed-loop system. We say that the system is in deadlock if *all* of its components are in a deadlock state. The following TRIO formula captures this notion of deadlock: it is true if all components $c \in C$ (with C the set of system components) can reach a deadlock state:

$$\bigwedge_{c \in C} \bigvee_{x \in dom(S_c)} \text{SomF}\left(\text{AlwF}\left(S_c = x\right)\right) \tag{14}$$

where $dom(S_c)$ is the set of states of component c, SomF and AlwF are the TRIO counterparts of the **eventually** and **globally** LTL operators; they are defined as follows:

$$\text{SomF}(F) \overset{def}{=} \text{Until}(true, F)$$
$$\text{AlwF}(F) \overset{def}{=} \neg\text{SomF}(\neg F)\,.$$

Finally, we discuss a property concerning the possibility to produce and deliver one processed workpiece of any kind within T time units from the system startup. The property is captured by the following formula:

$$\text{WithinF}((S_{Rob} = GoToCo1) \vee (S_{Rob} = GoToCo2), T) \tag{15}$$

The formula states that, within T time units from the start of the system, one of the two states $GoToCo1$ or $GoToCo2$ of Fig. 4 is reachable. The Stateflow diagram reaches state

Table 2 Test results

Formula	Time (s)	Memory (Mb)	Result
Zeno paths detection (13)	85	264	No
Deadlock detection (14)	17,991	268	No
Workpiece, T = 15 (15)	407	260	No
Workpiece, T = 20 (15)	89	272	Yes

$GoToCo1$ if a workpiece of any type has been produced by *Machine 1*, and similarly for $GoToCo2$ and *Machine 2*. WithinF is a TRIO operator derived from Dist:

$$\mathrm{WithinF}(F, T) \overset{def}{=} \bigvee_{0 \le t \le T} \mathrm{Dist}(F, T) .$$

In our tests, we have used two values, 15 and 20, for T. Formula (15) does not hold if $T = 15$, but it does if $T = 20$. By analyzing the output of the ℤot tool in the latter case we found that 16 is the minimum number of time units to satisfy the formula (which can be confirmed by checking formula (15) with $T = 16$).

Some performance results obtained during the verification of properties (13) and (14–15) with the two values of T mentioned above are shown in Table 2. In all cases ℤot has been set up with a bound for the length of analyzed runs equal to 70. Table 2 shows the time required by the tool to check the property, the memory occupation and the result, i.e. whether the property holds or not. All tests have been carried out on a 3.3 Ghz quad core PC with 4 Gbytes of Ram.

The Stateflow diagram of Fig. 4 has $12 \cdot 2^{18}$ possible configurations (corresponding to the state space of cardinality $|S| \cdot 2^{|D|}$); the overall system model, which also includes diagrams for all the other components, is considerably larger. As a consequence, deadlock detection analysis (formula (14)) takes a long time, as the tool must exhaustively analyze all possible runs. Formulae (13) and (15), instead, formalize *reachability* properties; their analysis is much faster, since the tool stops as soon as it finds a run that satisfies the formula.

To conclude this section, we briefly illustrate an example of verification that allowed us to detect and correct errors in a previous version of the model. By feeding ℤot formula (14) on an earlier model of the robotic cell, the tool determined that deadlock configurations did exist, and it returned a case of deadlocked run. By studying this run, we discovered that the system model remained forever in configurations with state $GoToCIn1$, (see Fig. 4). The run, which is summarized in Fig. 9, shows a problem in the communication protocol between the *Robot* component and the *Machine1* component, which also affects the cell *Controller*.

The run is presented as a timeline, starting from 0. Over each micro-step we report the partial configuration with the

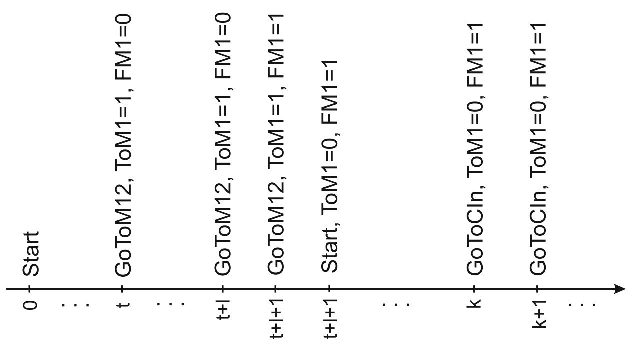


Fig. 9 Deadlocked run returned by ℤot

name of the active state of the *Controller* component of Fig. 4, and the values of input variable $FM1$ and output variable $ToM1$ (the other variables are not shown). The label below the micro-step (e.g., t) identifies the macro-step to which it belongs. When the *Controller* component enters state $GoToM12$ at time t, it signals to *Machine 1*, through an output event modeled by variable $ToM1$, that the *Robot* has just arrived. After working for l instants of time, at time $t + l + 1$ *Machine1* signals to the *Controller* the "work termination" event, which is mapped to variable $FM1$. The transition between states $GoToM12$ and $Start$ becomes enabled and the *Controller* component returns to the initial state, resetting $FM1$ to signal the end of the communication. The problem occurs if the *Controller* component reaches state $GoToCIn$ a second time, as at time k. In fact, from the time instant $k+1$, the *Controller* component cannot leave state $GoToCIn$ anymore since variable $FM1$ has not been reset by *Machine1*. Hence, no outgoing transitions are enabled. After correcting the error, a new check of property (14) showed that the modified system model is deadlock-free.

Conclusions

We presented an approach to the formal verification of control designs for FMSs based on Stateflow diagrams and temporal logic. The approach, which has been implemented in the ℤot verification tool, allowed us to check significant properties of an example FMS, and to unearth an error in an earlier design of the controller. In particular, the technique presented in this paper allows designers to check the model for the presence of so-called Zeno runs, which are unwanted behaviors whose detection is not possible with existing automated tools, even those that are not specific for FMS (Esteve et al. 2012). In addition, our logic-based approach facilitates a rather fine analysis of the temporal behavior of Stateflow diagrams, as it permits users to separate between and predicate on different micro-steps of the same macro-step (e.g., the first vs. the last micro-step of the same macro-step). One of the strengths of our approach is also that it allows designers to create models that can be formally verified with automated tools starting from domain-specific notations such as, for example, Stateflow diagrams. This is a necessary step towards the goal of allowing domain experts that are well-versed in the design of FMSs, but which have little familiarity with formal verification techniques, to carry out formal verification activities on the models of their choice, rather than those imposed by the underlying tool.

Future work will focus on creating a complete environment through which FMS experts can seamlessly move from the modeling to the verification of their designs, then receive feedback from the formal verification tool without having to directly access the formal concepts underlying the environment. The prototype formal verification tool that supports the technique presented in this paper shows the feasibility of our approach. In the future we will study mechanisms for improving the efficiency of the verification phase, in particular by exploiting the hierarchical and modular nature of the analyzed Stateflow/Simulink diagrams. We will also explore the application of our approach to other industry standards besides IEC 61499.

References

Alur, R., & Henzinger, T. (1999). Reactive modules. *Formal Methods in System Design*, *15*, 7–48.

Ballarino, A., & Carpanzano, E. (2002). Modular automation systems design using the IEC 61499 standard and the simulink/stateflow toolboxes. In *Proceedings of the asme Japan–USA symposium on flexible automation*.

Basile, F., Chiacchio, P., Vittorini, V., & Mazzocca, N. (2004). Modeling and logic controller specification of flexible manufacturing systems using behavioral traces and petri net building blocks. *Journal of Intelligent Manufacturing*, *15*, 351–371.

Bersani, M., Frigeri, A., Morzenti, A., Pradella, M., Rossi, M., & San-Pietro, P. (2010) Bounded reachability for temporal logic over constraint systems. In *Proceedings of the 17th international symposium on temporal representation and reasoning (TIME)*, pp 43–50.

Brusaferri, A., Ballarino, A., & Capanzano, E. (2011). Reconfigurable knowledge-based control solutions for responsive manufacturing systems. *Studies in Informatics and Control (SIC)*, *20*, 31–42.

Ciapessoni, E., Crivelli, E., Coen-Porisini, A., Mandrioli, D., Mirandola, P., & Morzenti, A. (1999). From formal models to formally-based methods: An industrial experience. *ACM Transactions on Software Engineering and Methodology*, pp. 79–113.

Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, et al. (2002). NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th internationl conference on computer aided verification (CAV)*, pp. 359–364.

Esteve, M. A., Katoen, J. P., Nguyen, V. Y., Postma, B., & Yushtein, Y. (2012). Formal correctness, safety, dependability, and performance analysis of a satellite. In *Proceedings of the international conference on software engineering (ICSE)*, pp. 1022–1031.

Furia, C. A., Mandrioli, D., Morzenti, A., & Rossi, M. (2012). *Modeling time in computing*. EATCS Monographs in Theoretical Computer Science. Berlin: Springer.

Gourcuff, V., DeSmet, O., & Faure, J. (2008). Improving large sized plc programs verification using abstractions. In *Proceedings of the 17th IFAC world congress*.

Hanisch, H. M., Lobov, A., Lastra, J. M., Tuokko, R., & Vyatkin, V. (2006). Formal validation of intelligent-automated production systems: towards industrial applications. *International Journal of Manufacturing Technology and Management, 8*(1), 75–106.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, *8*(3), 231–274.

Harel, D., & Naamad, A. (1996). The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, *5*(4), 293–333.

IEC. (2003). International Standard IEC61131-3, Programming Languages for Programmable Controllers. International Electro-technical Commission, (IEC), 2nd edn.

IEC. (2005). International Standard IEC61499, Function Blocks, Part 1–4. International Electro-technical Commission, (IEC), 1st edn.

ISaGRAF IT. (2012). Isagraf6 developer web site and online documentation. http://www.isagraf.com.

Khalgui, M., Mosbahi, O., Hanisch, H. M., & Li, Z. (2012). A multi-agent architectural solution for coherent distributed reconfigurations of function blocks. *Journal of Intelligent Manufacturing*, *23*, 2531–2549.

Klein, S., Weng, X., Frey, G., Lesage, J., & Litz, L. (2002). *Controller design for an FMS using signal interpreted Petri nets and SFC*. In *Proceedings of the American control conference*, pp. 4141–4146.

Levi, F. (2000). Compositional verification of quantitative properties of statecharts. *Journal of Logic and Computation*, *11*(6), 829–878.

Lewis, R. (2001). Modelling control systems using iec 61499. applying function blocks to distributed systems. IEEE Publishing.

Mathworks. (2011). Stateflow online documentation. http://www.mathworks.it/help/toolbox/stateflow/.

Mazzolini, M., Brusaferri, A., Carpanzano, E. (2010). Model-checking based verification approach for advanced industrial automation solutions. In *Proceedings of the international conference on emerging technologies and factory automation*, pp 1–8.

Pradella, M., Morzenti, A., & San Pietro, P. (2008). Refining real-time system specifications through bounded model- and satisfiability-checking. In *Proceedings of the 23rd IEEE/ACM international conference on automated software engineering*, pp. 119–127.

Pranevicius, H. (1998). Formal specification and analysis of distributed systems. *Journal of Intelligent Manufacturing*, *9*, 559–569.

Thapa, D., Park, C., Dangol, S., & Wang, G. (2006). III-phase verification and validation of IEC standard programmable logic controller. In *Proceedings of the IEEE international conference on computational intelligence for modelling control and automation*, pp. 111–111.

Vyatkin, V. (2011). IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics*, *7*(4), 768–781.

Vyatkin, V., & Hanisch, H. M. (2003). Verification of distributed control systems in intelligent manufacturing. *Journal of Intelligent Manufacturing*, *14*, 123–136.

Vyatkin, V., Hanisch, H. M., & Pfeiffer, T. (2003). Object-oriented modular place/transition formalism for systematic modeling and validation of industrial automation systems. In *Proceedings of the IEEE international conference on industrial informatics*, pp. 224–232.

Wang, J., & Deng, Y. (1999). Incremental modeling and verification of flexible manufacturing systems. *Journal of Intelligent Manufacturing*, *10*, 485–502.

Zhang, D., & Anosike, A. (2012). Modelling and simulation of dynamically integrated manufacturing systems. *Journal of Intelligent Manufacturing*, *23*, 2367–2382.