Proceedings of the 14th IFAC Symposium on Information Control Problems in Manufacturing Bucharest, Romania, May 23-25, 2012

Automated Formal Verification for Flexible Manufacturing Systems

Emanuele Carpanzano ** **Luca Ferrucci** * **Dino Mandrioli** *
Mauro Mazzolini ** **Angelo Morzenti** * **Matteo Rossi** *

* *Politecnico di Milano, Dipartimento di Elettronica e Informazione, Piazza L. Da Vinci 32, 20133 Milano, Italy (e-mail: {ferrucci,mandriol,morzenti,rossi}@ elet.polimi.it).*
** *CNR-ITIA, Via Bassini 15, 20133 Milan, Italy (e-mail: {emanuele.carpanzano,mauro.mazzolini}@itia.cnr.it)*

Abstract: In this paper we present an effective approach to perform formal verification of properties of interest of production systems whose behavior is modeled through Stateflow diagrams. The approach hinges on a semantics of Stateflow diagrams given in terms of formulae of a metric temporal logic. The semantics has been implemented in a fully automated tool which allows users to define a wide range of properties of interest and then to check whether they hold for the system or not. We also point out an error in the design of the system that has been detected by applying the technique.

Keywords: Formal methods, formal verification, bounded model checking.

1. INTRODUCTION

Being able to precisely analyze the properties of control systems of manufacturing plants before their deployment on the field is of the utmost importance to reduce both the duration and the costs of the deployment phase. To this end, various techniques have been defined over the years. Many of these are based on simulation mechanisms, where a fixed, usually large, number of executions of a model of the system being designed are generated and then analyzed, e.g. to estimate performance parameters of the plant (Ballarino et al. (2009)). More recently, approaches based on formal verification mechanisms, which are able to exhaustively explore the execution space of a system model, have been studied and proposed for the design of manufacturing systems. 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 notation to those of the other is often difficult.

In this paper we present a formal verification technique for models of manufacturing systems whose main ingredients are: (i) Stateflow diagrams as the notation for the modeling of the behavior of designed systems; (ii) a semantics of Stateflow diagrams based on a decidable metric temporal logic; (iii) a fully automated 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 chosen as an example to explain the approach, other formalisms (e.g. Timed automata, SFC, Petri nets, etc. . .) 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: Section 2 frames this work in the context of existing techniques, highlighting the features that separate it from them; Section 3 illustrates our approach to the verification of control designs for manufacturing systems, applies it to an example system, and presents some experimental results of the verification of the example system, with emphasis on the design errors unearthed through the analysis; Section 4 concludes.

2. FORMAL METHODS FOR THE VERIFICATION OF AUTOMATION SOLUTIONS

Verification is the process of checking the robustness and reliability of the designed control solution by proving its compliance with a given specification. During the last few years, many research efforts have been focusing on exploring and developing new methodologies supporting the verification process through formal approaches. Different types of formal models, as well as logics for the definition of the properties to be proved for the model have been investigated. In (Klein et al. (2002)) the model of the control system is defined in terms of Signal Interpreted Petri Nets; these models are verified using a symbolic model checking tool, and are then translated into the IEC 61131-SFC language. (Vyatkin et al. (2003)) develop a formal model of automation solutions based on Net Condition/Event Sys-

978-3-902661-98-2/12/$20.00 © 2012 IFAC

10.3182/20120523-3-RO-2023.00133

tems (NCES); models are analyzed through SESA model checking and the properties are defined through temporal logic. In (Mazzolini et al. (2010)) Stateflow diagrams are used as the modeling notation, model coverage properties proposed by the DO 178B standard are considered as properties to be proved, and Simulink Design Verifier is adopted as formal verification tool. In (Gourcuff et al. (2008)) a representation of logic controllers programs aiming at improving scalability of model-checking techniques within the industrial automation domain is proposed. The benefits of this representation are shown by means of three examples using NuSMV as model checking tool. In (Thapa et al. (2006)) the developed PLC code is translated 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 fairly wide community of practicioners; our approach, however can be easily adapted to any state-based, possibly graphical, notation according to the preferences of the selected user comunity (ii) a formal semantics of the Stateflow-based notation given in terms of a metric temporal logic; and (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. The next section provides some details of the proposed approach and highlights its main benefits.

3. METHODOLOGY OVERVIEW & CASE STUDY

The main ingredients of the approach presented in this paper are the TRIO metric temporal logic (Ciapessoni et al. (1999)) for the definition of both the semantics of Stateflow diagrams and the properties to be checked, and the Zot bounded satisfiability checker (Pradella et al. (2007)) for the automated formal verification of TRIO models. TRIO is a first-order linear temporal logic with a metric notion of time, which allows users to state properties such as "formula F will hold 5 time units in the future from the current instant". In addition to the metric temporal operators, TRIO supports the typical qualitative LTL operators such as "Until", "Always", "Eventually". TRIO supports both continuous and discrete time domains, though in this work we focus on the latter. Zot is a tool that accepts as input models described through various linear temporal logics (with or without a metric notion of time). These models are essentially sets of constraints defining all acceptable traces of the described system. Given a model M, Zot determines whether it can be satisfied (i.e., it has at least an acceptable trace) or not and, in the former case, it returns one such trace. Later in this section we describe how we use the mechanisms implemented in Zot to verify user-defined properties of the systems under development; it is possible, however, to use different tools that are capable to verify LTL specifications, such as the symbolic model checker Nu-SMV2, which can be connected to the off-the-shelf Minisat SAT Solver and/or ZChaff SAT Solver, as we can see in (Cimatti (2002)).

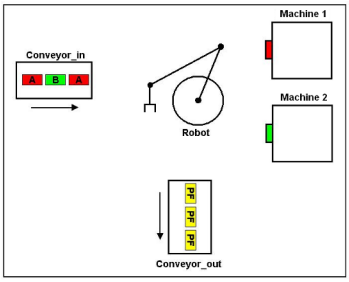

Fig. 1. Robotic Cell.

We present our approach through an example of a robotic cell of a Flexible Manufacturing System (FMS). The model is composed of a robot arm that performs loading and unloading of parts of different types on two machines. The cell, as shown in Figure 1, is served by a conveyor belt (*Conveyor_in*), which provides pallets to be processed. There are two types of pallets; those of type A must be processed by *Machine 1* while those of type B are processed by *Machine 2*. The finished parts are discharged from the cell by the conveyor out belt (*Conveyor_out*).

In FMS systems non-deterministic choices within single components must be avoided. In our example, in case of multiple requests from different components, we obtain this by statically assigning priorities to the operations performed by the robot. More precisely, the unloading of workpieces from the machines has higher priority than their loading and unloading pieces from *Machine 1* has precedence over unloading from *Machine 2*. Finally, the robot arm can switch at any point in time 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 upon a suitable command.

To describe the control logic of each component of the FMS we use Stateflow diagrams. For example, the Stateflow diagram of Figure 2 provides a model of the behavior of component *Robot* of Figure 1. Diagrams for all remaining components of the robotic cell have also been defined, but they are not shown here due to lack of space. [1]

The Stateflow notation is a variation of Statecharts (Harel (1987)). In a nutshell, a Stateflow diagram 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 that are used to represent input and output events: a variable v_i (resp. v_o)

[1] See `home.dei.polimi.it/rossi/FMS_TR.pdf` for further details.

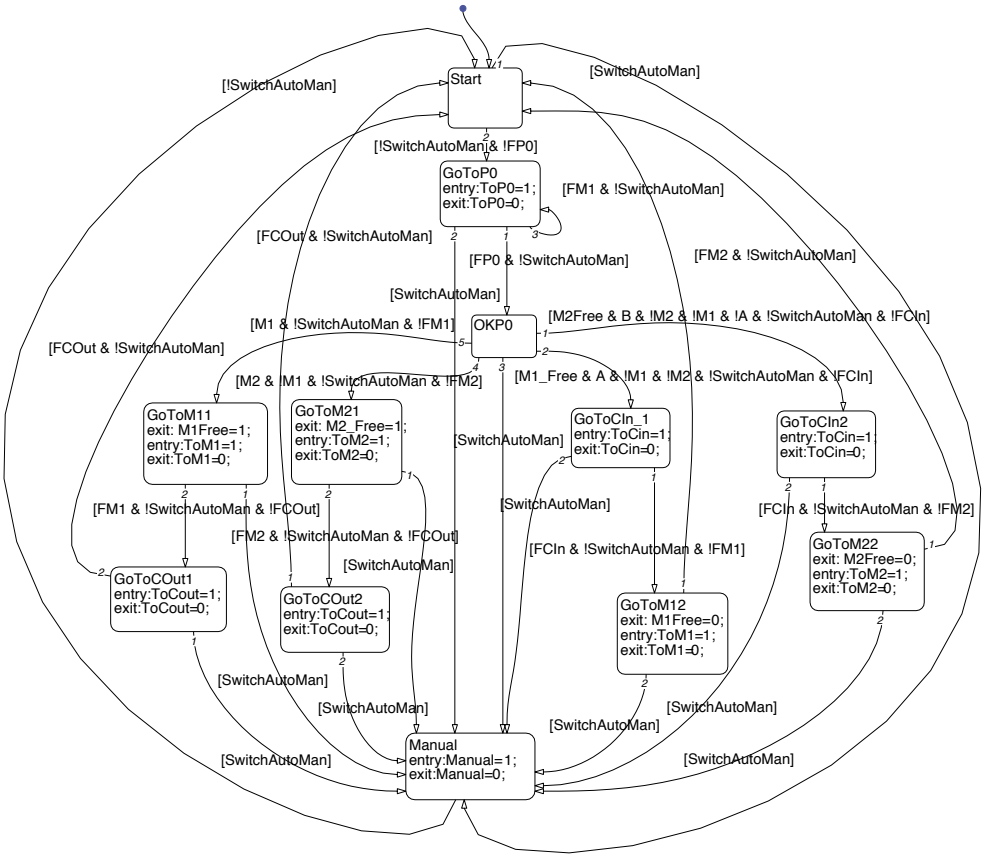

Fig. 2. Stateflow diagram of the controller of the robotic cell of Figure 1.

modeling an input (resp. output) event is set to *true* in those instants 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 throughout 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 a non-input variable.

(3) A finite set of transitions, T, that may include guards (i.e., constraints) on the variables of D and actions.

(Mathworks (2011)) presents the complete, informal, specification of Stateflow diagrams. The next sections outline the salient parts of our formal semantics of the notation.

3.1 Semantics

Our semantics of Stateflow diagrams is based on the STATEMATE semantics of Statecharts (Harel and dNaamad (1996)). The semantics includes a composition operator for building hierarchical, modular models from simpler ones. Due to the lack of space, in this paper we focus on the semantics of single components [2] .

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 $\mu_i : D \to dom(D)$. Expressions on the variables of D are evaluated in the active configuration. A transition $s \xrightarrow{g/a} s'$ 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, which entails that there cannot be more than one transition enabled in the same configuration.

The semantics of the evolution of time in Statecharts (hence, in Stateflow) has proven difficult to pin down precisely, and different solutions have been proposed (e.g., (Levi (2000); Alur and Henzinger (1999))). Our model is of the so-called **run-to-completion** variety. In this model the system reacts to the input events by performing a sequence of reactions 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. Micro-steps take zero time to execute; when no more 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

[2] We refer the reader to `http://home.dei.polimi.it/rossi/FMS_TR.pdf` for additional details on diagram composition

of macro-steps. In summary, the semantics of a macro-step is as follows:

(1) When macro-step begins, input data and events are assigned to the corresponding variables of set D_I.
(2) As long as there are enabled transitions, micro-steps are executed in zero time.
(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 of one unit, and output events and data produced during the macro-step are communicated to the environment.

Thus each 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 actual 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 that its reaction to inputs I_i is completed before inputs I_{i+1} are produced.

The definition of step in Statecharts (and Statecharts-like notations) has long been debated in 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 behavior 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 hits a loop of micro-steps that is never exited, thus never triggering the advancement of time. Zeno runs must be avoided in models because they represent unfeasible behaviors. In section 3.3 we will show how Zeno runs can be detected using our formal semantics of Stateflow.

3.2 Temporal Logic Encoding

This section presents the formalization of the semantics of Stateflow diagrams using temporal logic. We use the qualitative temporal operators of LTL to describe the succession of micro-steps, and the metric operators specific of TRIO to express quantitative properties of macro-steps. Ultimately, the semantics is implemented in the input language of the Zot tool, thus allowing users to perform automatic verification of Stateflow models.

For each variable $V \in D$ of the Stateflow model we introduce a corresponding Zot variable with domain $dom(V)$. When groups of variables have the same domain, we group then in Zot *arrays* (i.e., sequences of variables that are accessed through an index). In the case of the controller of the robotic cell of Figure 2 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 Zot 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 Figure 2, $StateCRO$ 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 guard g_i, we introduce the following formula:

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

where $\bigcirc$ is the usual LTL **next** operator, G_i is a Boolean formula encoding guard g, 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 G_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:

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

where N is the number of transitions of the diagram, and subformula $NOCHANGE$ asserts that in the next micro-step the current state and the values of all output and local variables do not change. Subformula $NOCHANGE$ is not detailed here for space reasons. 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:

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

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; additional predicates and formulae, not shown here for the sake of brevity, are introduced in the composed system model for the synchronization of the local ticks.

Now, we introduce a formula asserting that when predicate *tick* is false, then the values of the input variables D_I must be the same as those in the preceding micro-step:

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

where the $\overleftarrow{\bigcirc}$ and $\Box$ are, respectively, the **yesterday** and **globally** LTL operators. More precisely, $\overleftarrow{\bigcirc}F$ holds

if formula F held the previous micro-step, while $\Box F$ holds in the current and in all future micro-steps. The conjunction of formulae $\bigwedge_{i=1}^{N}(1)_i$, (2-4), is a formula SYS that holds only for all the possible runs represented by the Stateflow diagram, i.e. it encodes the behavior of the modeled system.

3.3 System properties verification and experimental results

In this section we use the encoding presented in Section 3.2 to check some typical relevant properties of the robotic cell M that is the composition of the 5 modules depicted in Figure 1.

First, we need to ensure 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:

$$\Diamond(\Box(\neg tick_M)) \tag{5}$$

where $tick_M$ is the global tick predicate of the robotic cell, and $\Diamond$ is the **eventually** LTL operator. Informally, $tick_M$ holds when all composing modules also tick; its full semantics is given in (Carpanzano et al. (2011)). $\Diamond F$ holds in a micro-step if there is a future micro-step in which formula F holds. Formula (5) holds if, from a certain micro-step on, the clock does not tick any more. We checked through the Zot tool that formula $SYS \wedge (5)$ is *unsatisfiable*, which means that there are no runs of the system that also show property (5), hence the system is devoid of 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. We first need to redefine the TRIO Dist operator, where $\mathrm{Dist}(F, K)$ holds in each instant t such that formula F holds at time $t + K$. The following formula defines the meaning of $\mathrm{Dist}(F, 1)$, to take into account that time advances only at the occurrence of *tick*, starting from the operators predicating on micro-steps:

$$\bigcirc(\neg tick \mathsf{U}(tick \wedge \bigcirc(\neg tick \mathsf{U}(\bigcirc tick \wedge F)))) \tag{6}$$

where U is the usual binary LTL operator **until**, and $A \mathsf{U} B$ holds in those micro-steps such that there is a future micro-step in which B holds, and A holds in all micro-steps up to that one (excluded). Formula (6) asserts that $\mathrm{Dist}(F, 1)$ holds if, at the end of the macro-step following the current one, formula F holds. The end of the next macro-step occurs the next micro-step when *tick* holds, without it being true in the micro-steps in between.

We also redefine the Until TRIO operator, which must predicate on macro-steps. More precisely, $\mathrm{Until}(A, B)$ is defined by the following formula:

$$\mathrm{Until}(A, B) = \bigcirc((\bigcirc tick \Rightarrow A) \mathsf{U} (\bigcirc tick \wedge B)) \tag{7}$$

Formula (7) asserts that $\mathrm{Until}(A, B)$ holds if there is a future micro-step 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, since that is when all system variables have certainly been updated for the current macro-step. $\mathrm{Dist}(F, K)$ is defined as $\mathrm{Dist}(\ldots \mathrm{Dist}(\mathrm{Dist}(F, 1), 1 \ldots, 1))$ (k times).

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:

$$\bigwedge_{c \in C} \bigvee_{x \in dom(S_c)} \mathrm{SomF}(\mathrm{AlwF}(S_c = x)) \tag{8}$$

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: $\mathrm{SomF}(F) \stackrel{def}{=} \mathrm{Until}(true, F)$, $\mathrm{AlwF}(F) \stackrel{def}{=} \neg\mathrm{SomF}(\neg F)$.

The last property we present in this paper is a behavioral property that states whether it is possible 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:

$$\mathrm{WithinF}((S_{Rob} = GoToCo1) \vee (S_{Rob} = GoToCo2), T) \tag{9}$$

The formula checks whether, within T time units from the start of the system, one of the two states $GoToCo1$ or $GoToCo2$ of Figure 2 is reachable. The Stateflow diagram reaches state $GoToCo1$ 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: $\mathrm{WithinF}(F, T) \stackrel{def}{=} \bigvee_{0 \le t \le T} \mathrm{Dist}(F, T)$.

In our tests, we have used two values for T: 15 and 20. With $T = 15$ formula (9) does not hold, whereas with $T = 20$ it does. 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 again formula (9) with $T = 16$). Some performance results obtained during the verification of properties (5) and (8-9) with the two values of T mentioned above are shown in Table 1. In all cases Zot has been set up with bound equal to 70 for all the formulas. This bound is the maximal length of runs analyzed by Zot and is a user-defined parameter of the verification. The table shows the time spent 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 PC with an Intel quad cores processor at 3.3 Ghz and 4 Gbytes of Ram.

The Stateflow diagram of Figure 2 has $12 \cdot 2^{18}$ possible configurations ($|S| \cdot 2^{|D|}$); the overall system model, which also includes diagrams for all the other components, is

Table 1. Test results

Formula	Time (sec)	Memory (Mb)	Result
Zeno Paths detection (5)	85	264	No
Deadlock detection (8)	17991	268	No
Workpiece, T=15 (9)	407	260	No
Workpiece, T=20 (9)	89	272	Yes

even bigger. As a consequence, deadlock detection analysis (formula (8)) takes a considerable amount of time since the tool must analyze all possible runs exhaustively.

Formulae (5) and (9) are so-called *reachability* formulae, so the analysis in this case is much faster, since the tool stops as soon as it finds a run that satisfies the formula. Finally, notice that memory consumption is rather low since it depends only on the size of the model.

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 (8) on an earlier model of the robotic cell, the tool determined that deadlocks 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 Figure 2). The run, which is summarized in Figure 3, shows a problem in the communication protocol between the *Robot* component and the *Machine 1* component, which also affects the cell *Controller*.

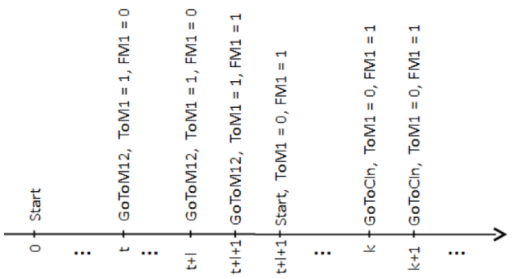

Fig. 3. Deadlocked run returned by Zot.

The run is presented as a timeline, starting from 0. Over each micro-step we report the partial configuration with the name of the active state of the *Controller* component of Figure 2, 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 $t + l + 1$ *Machine 1* 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 shown 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 *Machine 1*. Hence, no outgoing transitions are enabled. After correcting the error, a new check of property (8) showed that the modified system model is deadlock-free.

4. CONCLUSIONS

In this paper 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. Future work will focus on improving the efficiency of the verification phase. Also, we will explore applying our approach to other industry standards such as IEC 61499.

REFERENCES

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

Ballarino, A., Brusaferri, A., and Carpanzano, E. (2009). Adaptive automation solutions for responsive manufacturing systems. In *Proc. of CARV*, 10 pages.

Carpanzano, E., Ferrucci, L., Mandrioli, D., Mazzolini, M., Morzenti, A., and Rossi, M. (2011). Automated formal verification of flexible manufacturing systems. Technical report. `http://home.dei.polimi.it/rossi/StateflowZotSemantics\_TR.pdf`.

Ciapessoni, C., Mirandola, P., Coen-Porisini, A., Mandrioli, D., and Morzenti, A. (1999). From formal models to formally-based methods: an industrial experience. *ACM TOSEM*, 79–113.

Cimatti, A. (2002). Nusmv 2 : An opensource tool for symbolic model checking. In *Proc. of 14th Conf. on Computer Aided Verification*, 359–364.

Gourcuff, V., DeSmet, O., and Faure, J. (2008). Improving large sized plc programs verification using abstractions. In *Proc. of the 17th IFAC World Congress*.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Sci. of Comp. Prog.*, 8(3), 231–274.

Harel, D. and dNaamad, A. (1996). The STATEMATE semantics of statecharts. *ACM TOSEM*, 5(4), 293–333.

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

Levi, F. (2000). Compositional verification of quantitative properties of statecharts. *J. Log. Comp.*, 11(6), 829–878.

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

Mazzolini, M., Brusaferri, A., and Carpanzano, E. (2010). Model-checking based verification approach for advanced industrial automation solutions. In *Proc. of EFTA*, 8 pages.

Pradella, M., Morzenti, A., and SanPietro, P. (2007). The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In *Proc. of ESEC/SIGSOFT FSE*, 312–320.

Thapa, D., Park, C., Dangol, S., and Wang, G. (2006). III-phase verification and validation of IEC standard programmable logic controller. In *Proc. of IEEE Int. Conf. on Comp. Int. for Mod. Cont. and Aut.*, 111–111.

Vyatkin, V., Hanish, H., and Pfeiffer, T. (2003). Object-oriented modular place/transition formalism for systematic modeling and validation of industrial automation systems. In *Proc. of IEEE INDIN*, 224–232.