Abstract—An important property of embedded systems is dependability. Today this addresses mostly safety and reliability. Guaranteeing these properties is normally done by adding redundancy to the system. This approach is expensive and can not cope with changing environments. Therefore new designs are researched, which allow systems to self-adapt and self-heal. For broad acceptance in industry it is important, that organic systems can be modeled and analyzed with standard modeling tools and languages.

We present a case study of an adaptive production automation cell modelled in the Lustre language using the SCADE Suite and the verification of functional properties. SCADE is used widely in industry, especially in safety critical applications. Being able to model and verify adaptive systems in SCADE could increase their acceptance for these target areas.

I. INTRODUCTION

Modern embedded systems are becoming increasingly complex. At the same time the risk associated with such systems and the cost of system failure is increasing steadily. This results in a strong demand for robust systems, which can compensate component failure and adapt to changing requirements. Several different approaches to solve this problem – like Autonomic Computing [1] or Organic Computing [2] – have been developed in the last years. The general idea is to allow the systems to react by themselves to changing conditions, giving them a certain degree of freedom for autonomic behaviour.

In many of these approaches nature-inspired methods like genetic-algorithms or ant-algorithms are used. These algorithms are used to mimic the behaviour of natural phenomena where simple local rules lead to complex global behaviour and disturbances are solved autonomously without external intervention. Using this principle, systems become able to cope with unforeseen changes in their environment, changes in their requirements or react to partial system failures [3].

On the other hand such methods may lead not only to intended behaviour but to unwanted behaviour as well. Due to the nature of the algorithms, it is very hard to predict [3] which types of reactions the system will show. But giving (safety) guarantees about a system is crucial for certain applications like automotive, avionics and also production automation. Therefore it is very important to analyze such systems very precisely. SCADE and Lustre are widely used in these areas for formal analysis and verification [4]. We show how the SCADE design verifier may be used to rigorously analyze organic SCADE models. We illustrate the process on a real-world example from production automation.

II. CASE STUDY

The case study describes an automated production cell with three robots, which are connected with autonomous transportation units. Every robot can accomplish three tasks: drilling a hole in a workpiece, inserting a screw into this hole and tightening the screw. These tasks are accomplished with three different tools that can be switched. Every workpiece must be processed by all three tools in a given order (first drill, then insert and finally tighten = DIT). Workpieces are transported between the robots by autonomous transportation units (carts). Changing the tool of a robot requires some time. Therefore the standard configuration of the system is that the three tasks are spread out between the three robots and the carts transfer workpieces accordingly. This situation is shown in Fig. 1.

Fig. 1. Valid configuration of robot cell

We examine the case when one or more tools break and the current configuration allows no more correct DIT processing of the incoming workpieces. If the drill of robot 1 breaks then DIT processing is no longer possible, as no other robot is configured to drill. A non-adaptive production cell would now come to a standstill and wait for maintenance.

This can be resolved like shown in Fig. 2. For this error resolution, not only the assignment of the tasks to the robots must be changed, but also the routes of the carts. If only the tools were switched, the processing of all tasks would be possible, but not in the correct order.

This reconfiguration was easy to do. A reconfiguration algorithm seems easy to design and analyze. But what will happen if several errors occur over time, triggering several reconfiguration steps? It is even possible, that during production unused tools of the robots are repaired by maintenance. If all these effects are taken into account it soon becomes very hard to design a reconfiguration algorithm. Once such an algorithm
III. LUSTRE AND SCADE

Lustre is a typed synchronous dataflow language with a discrete time model [5]. A dataflow is a sequence of pairs \((v, \tau)\) with \(v\) being a value with a type. The time step \(\tau\), is an instant of a sequence of natural numbers giving the discrete time step of a given value [6]. A flow needs not to be available at every time step of the run of a system, a pair of \((v, \tau)\) with \(\tau = 0, 2, 4, \ldots\) would correspond to a flow that is available only at every even time step.

The type of a flow can be either an atomic type of \(\text{bool, int, real}\) or a user defined structure composed of atomic types. Dataflows can be combined by arithmetic operations, conditionals and boolean functions, depending on their type. Other operators are for example \(\text{pre}\) and \(\rightarrow\). The \(\text{pre}(x)\) accesses the value of the flow \(x\) in the last time step. To initialise a flow, the \(\rightarrow\) operator is used.

Lustre does not allow recursion, i.e. a flow \(x\) may not be defined via a function dependent on the value of the flow \(x\). Feedback loops can be constructed using the \(\text{pre}\) operator, e.g. \(x = c \rightarrow \text{pre}(x) + d\) creates a flow \(x\) that is calculated from the previous value of \(x\) plus the constant \(d\). Its initial value \(\rightarrow\) the value of the time step \(0\) – is \(c\).

To create Lustre programs, the graphical SCADE tool of Esterel Technologies is available [6]. It allows using Lustre language features as graphical operators, create user-defined operators and design hierarchical Lustre models. It also allows to create state charts, called “safe state machines” (SSM). SCADE also has an integrated automatic model checker, the “design verifier” [7], that can prove safety properties by checking validity of given boolean flows for each instant on every possible trace of a system. Another interesting feature is the integrated code generator that is DO178B-level-A [6] certified.

These features make the SCADE/Lustre an interesting modeling environment for safety critical applications. It would be a great benefit to be able to express adaptive systems in this environment to increase their acceptance. One drawback is that there is no general possibility to express liveness properties like “the system always produces again in the future”.

IV. SCADE MODEL

The basic idea of modeling the adaptive production cell in SCADE is based on user defined operators for each robot and autonomous cart. The workpieces are modeled as a user defined dataflow with the structure shown in Table I.

|------------|---------------------------------------------------------------------------------------------------------------------|

The boolean flags of the workpiece dataflow are all \(\text{true}\) if a workpiece leaves the production cell and was processed in the right sequence. If \(\text{finished}\) is \(\text{false}\) then the workpiece is still being processed. The \(\text{finished}\) flag is set to true when the workpiece leaves the production cell. Modeling the workpieces as dataflow allows to express an unlimited number of workpieces, as there is always an incoming workpiece available from the workpiece dataflow. This is a clear improvement to the modeling in [8] where only a limited number of workpieces was modeled that were reset after processing.

The robots are modeled as SCADE operators with inputs for workpieces and for tools they should use. The output of the robot operator is a workpiece dataflow. The operator itself contains a SSM to simulate processing of the workpieces by changing the flags of the workpiece dataflow according to the configuration of the robot. Each task may take a different, predefined number of time steps to complete. This means the robots work independently of each other and of the autonomous carts, using the respective output and input ports for the workpieces as synchronisation points. This differs from [9] where a shared memory is used for data exchange.

The carts are modeled analogously to the robots. They also have an input for a workpiece dataflow and one for their configuration and an output dataflow for the workpieces.

The hierarchical structure of the model consists of four layers. The outermost layer is the whole production cell. Its inputs are the workpiece stream and the status of the robot tools. The output is the processed workpiece dataflow that can be used to define proof goals (see Sect. V). The next layer contains the input and output streams of the workpieces to and from the production cell, the controlling mechanism for the assignment of roles to the robots and the carts, as well as the interface to and from the robots and carts. Its basic layout is shown in Fig. 3. It also includes a feedback loop of the \(\text{toolStatus}\) from the robots to the control, as the control computes a new assignment of tools depending on the status of the respective tools. The following layer contains the two autonomous carts, the three robots and the available connections between them. The last layer contains the implementation of the carts and robots.

The configuration of the robots is done by the controlling mechanism via the \(\text{robotFunction}\) type, the carts are configured...
using the \textit{cartFunction} type (table II). For the robots this configures which tool to use, for the carts this configures which route to use for the workpieces. Failure of tools for the robots are modeled using the \textit{toolsStatus} (table II) type with one boolean flag for each tool of a specific robot. We use persistent failures models, i.e. if a tool fails once in the run of the system, it will not get available again in the same system run. We did not include other failures apart from the failure of tools but other failures can be integrated in the same way. Transient failures are possible as well.

\begin{verbatim}
type cartFunction = enum drillInsert, insertTighten
type robotFunction = enum drill, insert, tighten
type toolsStatus = {drillOk : bool, insertOk : bool, tightenOk : bool}
\end{verbatim}

\textbf{TABLE II} \hspace{1cm} \textbf{CART-, ROBOTFUNCTION AND TOOLSSTATUS TYPES}

The initial configuration of the carts and robots and the reconfigurations that are necessary because of failing tools of the robots are done by a reconfiguration algorithm that assigns a \textit{cartFunction} to each of the carts and a \textit{robotFunction} to each of the robots.

Fig. 4 shows the initial configuration of the system (black). For better readability only the flow of workpieces through the cell is shown. Possible reconfiguration scenarios are also shown (dashed).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig4}
\caption{Initial configuration of production cell}
\end{figure}

This algorithm is also modeled as a SCADE operator and is located within in the controlling mechanism. The challenge is to assure, that all tools are available and no assigned tool is broken and also that the carts are correctly assigned to the routes so that workpieces can be transported and processed in correct sequence. By using a SCADE operator it is possible to implement different reconfiguration algorithms either centralised or decentralised. This allows to easily compare different strategies for reconfiguration. It is even possible to use the assertion mechanism of SCADE to specify a general reconfiguration algorithm instead of implementing a specific one, analogously to our “restore-invariant” approach [8]. We implemented a simple algorithm that assigns a possible configuration by checking sequentially which tasks are possible for each robot.

\section{Verification}

Proof objectives in SCADE are boolean dataflows that are watched over every possible run of the system. If any run of the system is found that falsifies this dataflow, this run is given as a simulation run that allows to examine the behaviour of the system and to observe all dataflow values and internal variables. This run is then a counterexample to the assumed property.

An important property is that every workpiece that leaves the production cell is fully processed and processing was done in correct order. This can be checked by examining the outgoing workpiece dataflow, as the sequence of processing and which tasks have been conducted is stored in the workpiece data.

Another important property is that the production cell will finally produce even after a possible reconfiguration. This is a liveness property. Such properties can only be verified with a small workaround in SCADE. To do this we defined an “observing” SSM that counts how long no workpiece has left the production cell by observing the output dataflow. If this value is \textit{false} for too long, then the observer emits a \textit{false} boolean dataflow, thus falsifying its own safety property (see Fig. 5). In essence we do bounded checking of liveness properties, where the number of time steps is dependent on the system and must be adjusted accordingly. For our model with \(n\) failing tools, we used \(15 + 14n\) time steps for bounded checking of the liveness property. This number is the processing time for a workpiece in the production cell together with the time needed for a reconfiguration.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig5}
\caption{Observer automaton}
\end{figure}

Formally, we get the two safety properties \(SP_1\) and \(SP_2\) (eq. 1 and 2) as verification goals for the design verifier. Together these properties prove that workpieces are produced...
correctly and that workpieces will be produced. Both properties hold for the case study. This is not surprising, as no component failures have been taken into account.

\[
SP_1 := \text{finished} \rightarrow (\text{drilled} \land \text{inserted} \\
\land \text{tightened} \land \text{drilled1st} \\
\land \text{inserted2nd} \land \text{tightened3rd})
\]

(1)

\[
SP_2 := \neg \text{Observer}(\text{Error})
\]

(2)

The next step is to analyze the behaviour of the system wrt. component failures. As possible failures broken tools were modeled (failure of the carts have not been taken into account, but can be handled analogously). We use the assertion mechanism of the design verifier to guarantee that only selected tools may fail. Using this, we can analyze the system for critical combinations of failing tools that can lead to system failure in terms of violating the functional properties \(SP_1\) or \(SP_2\) as in [7]. The resulting sets of combination of failure modes are analogously to the minimal cuts sets of the widely used fault tree analysis (FTA) [10].

Formally we define a set \(\Delta\) of all \(\text{toolsStatus}\) variables of each robot and a set \(\Gamma \subseteq \Delta\) with the \(\text{toolsStatus}\) that may fail on a system run. This means all tools that have a corresponding element in the set \(\Gamma\) can fail on a system run, while the tools in the set \(\Delta \setminus \Gamma\) can not fail. Using this we can iteratively compute the critical combinations of failures, starting from single failures and then increasing the number of elements in the set \(\Gamma\). We use persistent failures but transient failures can be modeled in the same way. The only problem with transient failures is that \(SP_2\) may be false if a tool changes its status from working to failure and back, thus requiring a reconfiguration in every second time step. If this is possible, the cell is not long enough in a production phase and no workpieces are produced. Nevertheless under the assumption that the time between tool failures is long enough, it is possible to integrate transient failures.

To conduct verification we used the assertion mechanism of SCADE. We assert proposition 3 to indicate that the status of all tools that are not failing is true.

\[
\text{assert} \bigwedge_{t \in \Delta \setminus \text{fails}} t
\]

(3)

For the checking of the properties we used SCADE (Version 5.1 build 113) on an AMD 3500+ CPU with 1Gbyte RAM. Property \(SP_1\) needed from 46s (no failing tools) up to an average of 3h (3 failing tools) for verification. The liveness property \(SP_2\) needed from 44s (no failing tools) up to an average of 10h (2 failing tools). Unfortunately it was not possible to verify \(SP_2\) in reasonable time (less than one day) with more than two failing tools as at least the specified \(15 + 14n\) must be analyzed. Another reason for the long duration of the verification process may be that linear model checking has a higher complexity than branching time model checking [11] with for example computation tree logic (CTL). We were able to verify an analogous system in a matter of seconds with the Cadence SMV model checker [8] using CTL proof obligations. Therefore we would like to investigate transformation of SCADE models to SMV models as future work. Other topics of future work will be integrating semantically founded safety analysis techniques in SCADE, to generalize the modeling technique from our case study to more general form of Organic Computing systems and to apply this to different case studies.

VI. CONCLUSION

Our goal is to provide methods to guarantee behaviour of Organic Computing systems and are focusing on formal modeling and verification of these systems. To increase acceptance we want to integrate methods for this in tools and formalism that are widely used and accepted in industry.

We showed how to model a real-world example from production automation with self-x properties in SCADE and used the design verifier to verify functional properties of the cell, namely that workpieces are correctly produced and that they will finally be produced. We allowed tools of the robot to fail and a reconfiguration algorithm was integrated to bring the cell back to working mode. As only safety properties can be checked directly we defined an observer automation for checking a liveness property.

This approach for formalisation and verification of Organic Computing applications in SCADE should make it possible for developers using this tool to integrate self-x mechanisms into their projects getting the benefits of increased reliability and adaptation capabilities and still be able to formally verify their systems.

REFERENCES


