# Component-based Design and Analysis: A Case Study

Yan Jin Charles Lakos Robert Esser

*School of Computer Science, University of Adelaide, SA 5005, Australia*
{*yan, charles, esser*}*@cs.adelaide.edu.au*

## Abstract

*In this paper, we introduce a component-based design methodology and present a practical analysis approach that makes use of the modular nature of component-based designs to alleviate the state space explosion problem, a well-known obstacle to system verification. In addition, the approach is illustrated by application to a non-trivial case study: the production cell. It is shown that not only the basic consistency property, viz. the freedom from unexpected reception and deadlock, but also other important safety properties in the design can be proved.*

## 1. Introduction

In recent years, component-based software development has become more popular for the production of large-scale software applications. By building systems from independently developed components, a promising means of achieving software reuse, rapid development and complexity management is provided. However, as noted in [3]:

> *"system complexity, and hence the likely number of design errors, grows exponentially with the number of interacting system components"*.

This is a consequence of the well-known state space explosion problem and it largely limits the applicability of exhaustive analysis. To overcome this problem, various reduction techniques have been proposed in the literature. Among them, modular analysis (or compositional reasoning) [2, 7, 10, 11, 15, 20] is a powerful divide-and-conquer technique for decomposing the analysis task of a system into subtasks of individual components of the system. The key to this is to consider each component in conjunction with assumptions about the context of the component, and to consider the composition of components in conjunction with the interface behaviour of components.

In component-based systems, however, this key information is often missing or only informally described. Currently, the interface specifications of components tend to be rather restricted, capturing only the *signatures*, i.e. the names, data types and direction of information flow, but excluding information about the communication *protocols* of components. This is because software engineers lack a formal means for precisely specifying the interfaces behind which components encapsulate their services. As a result, components cannot be analysed independently due to the lack of information about the environments in which they are embedded, and the composition of components cannot be analysed due to the lack of rigorous specification of the interface behaviour of components.

We have developed a formal technique which focuses on communication protocols while abstracting away from the data values being communicated. The protocol of a component describes the services it provides, the way it reacts to its inputs and what it expects from its environment. It does not, however, disclose the implementation detail of the component. We use interface automata (IAs), a formal lightweight language proposed in [5], as the notation for describing protocols.

With the contextual assumptions captured by an interface automaton (IA), each component can be checked for conformance with the IA in isolation from the system. This ensures that a component is able to abide by the interaction protocol given by the IA, provided its environment behaves as expected. Furthermore, the composition of components can be analysed utilising the interface behaviour of components specified by the IAs while disregarding the internal activities of components. Using this divide-and-conquer approach, the state space explosion problem can be alleviated.

This paper focuses on the application of the above technique to a non-trivial case study. The formal presentation of this technique and comparison with related techniques is found elsewhere [14, 15]. The technique has been implemented in the context of the Moses tool suite [6], which presents an additional challenge to component-based development in that it supports the modelling and simulation of heterogeneous discrete-event systems, where components are modelled by different languages [6, 12, 13], e.g. process networks, Petri nets, Statecharts, etc.



Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**

The paper is structured as follows. In section 2, this research is compared with related work. Our approach to component-based design and analysis is presented in section 3 and is applied to the production cell case study in section 4. Finally, section 5 contains the conclusions.

It should be emphasised that throughout this paper the word *analysis* is used in the sense of verifying system properties rather than in any sense of requirements elicitation.

## 2. Related work

Many authors have contributed to the production cell case study, e.g. [16, 8, 9, 17]. Among these contributions, the closest to our approach is the work in [8, 9], where a detailed model of the production cell is found. Our design builds on this model but segments it into 7 loosely-coupled reusable components. Also, that work differs from ours in the analysis method employed. There, the analysis was directly conducted on the state space of the whole system with the help of reduction techniques such as stubborn set methods. In our work, the costly construction of the system state space is avoided. Instead, system properties are proved on the basis of independent model-checking of a group of small state spaces with the help of interface automata.

In [17], a design and verification approach to the production cell using UML is presented. There, components are modelled as UML Statecharts and the verification is conducted using the vUML tool which invokes the SPIN tool for executing the model checking task. As in [8, 9], this approach employs reduction techniques in order to explore all possible states of the system. This, however, trades time for memory. In contrast, our approach requires much less time and memory due to the smaller state spaces that need to be handled. Furthermore, our approach is not dedicated to a particular modelling language, but accommodates various notations for modelling components, including UML Statecharts [13].

In [16] there is a collection of other contributions and also a detailed comparative survey. To the best of our knowledge, there exists no other approach to the modular analysis of the production cell.

## 3. An approach to component-based design and analysis

When designing a system, an important step in decomposing the system into components is to specify the interaction protocols for components. These protocols serve as the contract between the system and components and guide component development. However, in the current practice of component-based development, software engineers often lack a formal means to capture protocols and instead use informal languages to express them. As a consequence, the correctness of component implementations is ensured only by the experience of the engineers, testing or some informal reasoning against the protocols. This approach is often error-prone with the design of the intermediate protocols providing no means of precise analysis.

By contrast, we introduce interface automata (IAs) [5] as a formal language that can describe the interaction protocol of every component at a high level of abstraction. The protocol includes not only how a component produces outputs in response to its inputs but also assumptions that the component makes on the environment as to when or what inputs are expected. The protocol, on the other hand, abstracts away the implementation details of the component, such as data values and internal behaviours. According to its protocol, a component can then be independently implemented and analysed.

The introduction of IAs breaks the system analysis task into a series of smaller tasks. Firstly, the conformance of each component with its associated IA is ensured, i.e. every component communicates with the environment in a way that conforms to the interaction protocol defined by its IA. Secondly, the consistency property of the system, viz. the freedom from deadlock and unexpected reception, can be determined by checking the compatibility of the interface automata. The compatibility refers to unspecified reception in an abstract system composed of these IAs. Because of the abstraction from components to their interface behaviour, the state space of the abstract system is much smaller than the component system. Hence the state space explosion problem is alleviated.

On the other hand, also due to the abstraction, safety properties dependent on the internal behaviour of components cannot be directly checked on the abstract system. We shall show that these properties can be proved by checking the state combinations of components deduced from the abstract system and individual component state spaces.

### 3.1. Interface automata

The IAs used here are deterministic finite state machines (FSMs) where input and output events are distinguished. The distinction reflects the fact that in asynchronous systems a component has control over its outputs but no control over its inputs. That is, a component decides when to produce an output, while the environment decides when an input event occurs.

Figure 1 shows an example IA, where bullets and arcs represent states and steps (or transitions), respectively. The state having an incoming arrow with no source denotes the initial state. Each transition is labelled by an event. For example, the IA has two input events "$a$" and "$b$" and an output event "$c$". This example IA specifies the interaction protocol for an adder component.



Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 IEEE

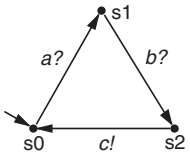



**Figure 1. An adder IA**

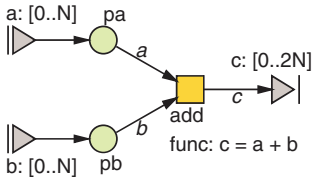



**Figure 2. An adder component**

The information contained in an IA is twofold. On the one hand, it restricts the output events that a component implementing it can produce in a particular state. On the other hand, it captures the designer's expectation that the environment should never provide an unspecified input event to the component. For example, figure 1 states that the environment can deliver $a$ and $b$ only once before $c$ is produced. Also, it guarantees that the component does not produce $c$ until both $a$ and $b$ have occurred.

### 3.2. Components

In this context, components are (finite) state machines assumed to communicate through input/output ports by discrete events. They consume data fed to input ports and produce data via output ports. An input/output event of the component refers to an occurrence of message transfer at one of its ports. Also, like [18, chapter 8], a component is required to be input-universal, i.e. it never refuses an input, so that writing to a component never blocks. This acknowledges the fact that components are often developed to function properly in unknown environments. This is also a requirement for independent deployment of components.

Let an IA event represent a class of component events associated with a particular port. Then we are able to use IAs to describe both the required interface behaviour and the environment assumptions of components. More specifically, an component event is considered to be a tuple $\langle e, v\rangle$ with $e$ an IA event (thus a component port) and $v$ a parameter (the data being communicated). We abbreviate such an event as $e.v$. The parameter can be arbitrary for a component input event, while for a component output event it is fixed and determined by the internal logic of the component. The association between IAs and components makes it possible to develop and analyse each component independently of other components and the ultimate context.

For example, we can develop an adder component or select one from the component library, according to the protocol specified by figure 1. Suppose we obtain the model shown in figure 2. The model is written in a variant of high-level Petri nets [12], where triangles represent the input and output ports and the body is given in the usual Petri net notation with circles, boxes and arcs representing places, transitions and the flow relationships, respectively. From the component perspective, an input port represents a (Petri net) transition of the environment which can fire only when expected by the IA and thus put a token into the connected place, while an output port represents a place of the environment for holding the tokens generated by the connected transition. The ports are typed with "$[0..N]$", meaning that in this example only tokens with integer values between $0$ and a constant $N$ can be transmitted via the ports.

### 3.3. Conformance checking of components

After a component is developed in this way, we need to make certain that the component does conform to the protocol specified by the IA. As the component is always assumed to run in a system where the input assumptions captured by the IA are respected, the conformance ensures that the component does not violate the output guarantees specified by the IA.

To do so, we first build the least helpful but adequate environment for the component from the IA specification. This is the environment that will not contradict the assumptions of the IA and such that any other suitable environment will be more helpful, i.e. it will supply less inputs to the component and will accept more outputs from the component. The environment also includes trap steps taken when a component violates the output guarantees specified by the IA, i.e. it produces an output event which is unspecified at a corresponding state of the IA. We refer to such an environment as the *most abstract implementation* of the mirror of the IA. For example, such an environment for the adder component is shown as a compact FSM in figure 3, where gray arcs represent trap steps, $\perp$ a single trap state, and $x, y, z$ integers in their respective domains.

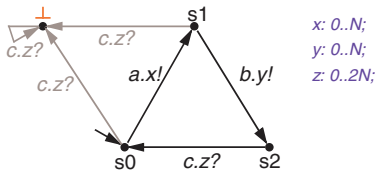



**Figure 3. The least helpful environment**

We then construct the *local state space* of the component paired with this least helpful environment. This will synchronise output events with input events between the com-



Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 IEEE

ponent and the environment. The states are tuples $\langle s, q\rangle$ with $s$ a IA state or $s = \perp$ and $q$ a component state, and the initial state is a tuple of the initial states of the IA and the component. We can now determine the conformance of the component by checking the absence of trap states, states $\langle s, q\rangle$ with $s = \perp$, and deadlocks in the local state space.

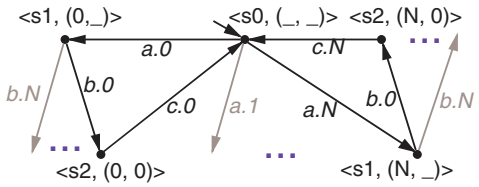



**Figure 4. Local state space of figure 2**

For example, the local state space of the adder component with respect to the adder IA is illustrated in figure 4. The initial state is $\langle s0, (\_, \_)\rangle$, where $s0$ is the initial state of the IA and $(\_, \_)$ is the marking of the component, indicating that there are no tokens in either place $pa$ or place $pb$. In this state, the environment is expected to provide an integer $v$ only via port $a$. As $v$ can lie between 0 and $N$, there are $N$ input steps emanating from the state, each receiving an event $a.v$ with distinct $v$. Take the step receiving $a.0$ as an example. The step ends at a state $\langle s1, (0, \_)\rangle$, where place $pa$ holds a token with value 0. Likewise, this state also has $N$ outgoing steps. If event $b.0$ is triggered, state $\langle s2, (0, 0)\rangle$ is entered, where both $pa$ and $pb$ hold a token of value 0. At this time, transition "add" in the component will be enabled and, if fired, will produce an output event $c.0$ with 0 representing the sum of tokens received via $a$ and $b$. The resultant state is the initial state. For brevity, we only expose a small portion of the local state space and omit the rest due to the structural similarity. As the local state space does not involve a trap state and is deadlock-free, we know that the adder component *conforms to* the adder IA.

### 3.4. Component-based designs

A typical component-based design process combines top-down and bottom-up design. Components are identified during system decomposition. IAs are then used to specify the interaction protocols for the components and to design the synchronisation patterns between components. The synchronisation patterns relate the output events to the input events of the IAs, while abstracting away the data being communicated. These resultant protocol specifications are then used for developing or selecting suitable components (or for further decomposition). Once the components have been developed or selected, they can then form a concrete *component-based design*, where the synchronisation patterns of IAs are reused for components with the incorporation of data values.

As an example, suppose two components, a user and an adder, are identified in a small adder system. We expect them to follow the protocols described in figures 1 and 6, respectively, and to synchronise as described in figure 5. In this case, an input event of one IA is synchronised with an output event of the other with the same name, e.g. "$a?$" of the adder with "$a!$" of the user. Using these protocols, we develop two components shown in figures 2 and 7, and connect them as shown in figure 5 to form the adder system. The user component randomly chooses two integers from place "nums" and provides them to the adder component for computation. Place "nums" initially holds all integers between 0 and $N$.

Since the IAs capture the interaction protocols expected by the designer of the components, we need to ensure that the components indeed follow these protocols in the execution of the ultimate system design. In other words, every input/output step of a component in the system corresponds to a step of its associated IA, ignoring the data being communicated. This property is called the *freedom from unexpected reception*. We also need to ensure that the system design is free from deadlock. A component-based design satisfying these is called *consistent*.

### 3.5. Analysis of system properties

The conformance of every component to its corresponding IA makes it possible to determine the system consistency property from an abstract system of IAs. The abstract system is an intermediate product we have obtained in an earlier design stage, and shares the same synchronisation patterns with the component-based system. In the system, an enabled output event of an IA is synchronised with all the input events of others related by a synchronisation pattern. The system is consistent if two conditions are satisfied. On the one hand, all constituent IAs are compatible, that is, no IA can produce an output event triggering an input event unspecified at the current state of of another IA. This is also called the *freedom from unspecified reception*. On the other hand, the abstract system is free from deadlock, that is, there is no deadlocked state in its state space (which is a FSM). It has been proved by the authors in [15] that the consistency of the abstract system can then serve as a sufficient condition for the consistency of the concrete system.

For instance, the IA system of the adder system consists of the IAs in figures 1 and 6 and has the synchronisation patterns shown in figure 5. Its state space looks like figure 1 except that all events are internal. It is easy to see that the IA system is consistent. Therefore so is the adder system. It should be noted that the example system is too small to demonstrate significant state space reduction using the proposed method.



Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**

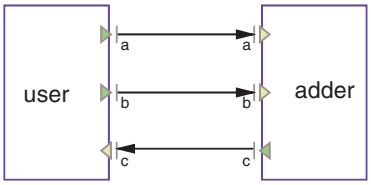



**Figure 5. An adder system**

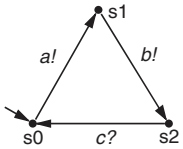



**Figure 6. A user IA**

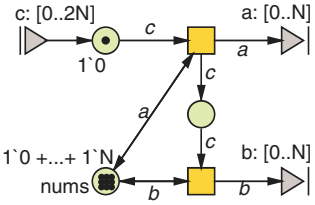



**Figure 7. A user component**

To determine safety properties other than the above, we have extended our work in [14, 15]. We have proved that local safety properties that are satisfied in the local state space of a component are also satisfied in a consistent system consisting of the component, as the assumptions captured in the specification IA are always respected in the system. These local properties are also called *component invariants*. The boundedness property of a system is a typical application of this theory, as it is the conjunction of the boundedness of all the components within the system, which is in turn a special component invariant. We have also justified that system safety properties can be ensured by proving the decomposition of the properties on the local state spaces of the involved components. We examine this in more detail in section 4.2.

### 3.6. Overview of the implementation

This approach is aimed at promoting a practical means for designing and analysing component-based systems. To achieve this, tool support is of great importance. Due to its extensibility, the Moses tool suite was chosen to be the implementation framework for the proposed methods.

The Moses tool suite addresses the definition of a certain class of visual languages and supports the visual modelling and simulation of discrete-event systems. A variety of notations can be used for describing components in such a system, e.g. Petri nets, Statecharts and process networks. This is made possible by Moses generic tools, such as a graph editor and a simulator, which are parametrised by visual language definitions.

On the basis of the Moses framework, we have encoded the formalism of IAs and developed tools for their composition and compatibility checking. We have also provided a tool for generating the local state space and thus the conformance checking of components coded in the pre-existing formalisms. In addition, we have integrated a small property specification language into the Moses so that safety properties can be specified. These include component invariants and system safety properties. Component invariants can be automatically checked using a newly-built state space exploration tool. As noted previously, this only requires the exploration of component local state spaces. The extensions for automatically checking more general system safety properties is currently under development.

The Moses tool suite is available at [1]. The tools implementing the techniques described in this paper are available in a side branch of the repository and will soon be merged into the main branch. In the meantime contact the authors for access.

## 4. Case study: the production cell

The production cell case study, posed in [16], was derived from a metal processing plant. The main task of the cell is to forge metal blanks in a press. The blanks are transported to and removed from the press through the collaboration of five other components in the cell: a feed belt, an elevating rotary table, a robot with two extendable arms, a deposit belt and a travelling crane. Figure 8 shows the top view of the cell (taken from [8]).

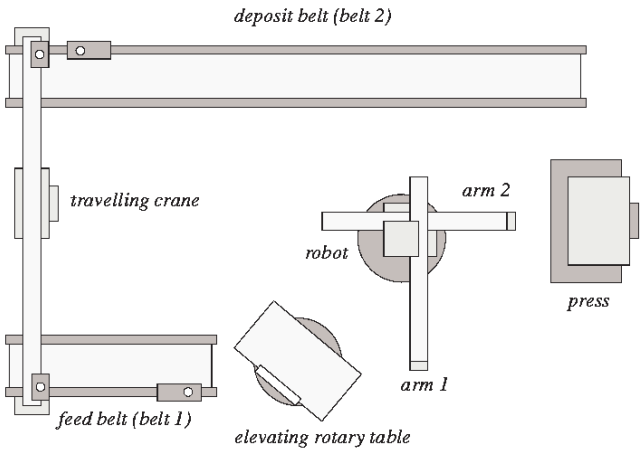



**Figure 8. Top view of the production cell**

The production cycle of a metal blank is as follows: When the feed belt conveys the blank to the table, the table rotates and lifts the blank to a position where the robot can pick it up using one of its arms. After the robot moves the blank to the press, the press forges the blank and then goes to a position where the robot can pick it up using the other arm. The robot then removes the pressed blank and



Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**

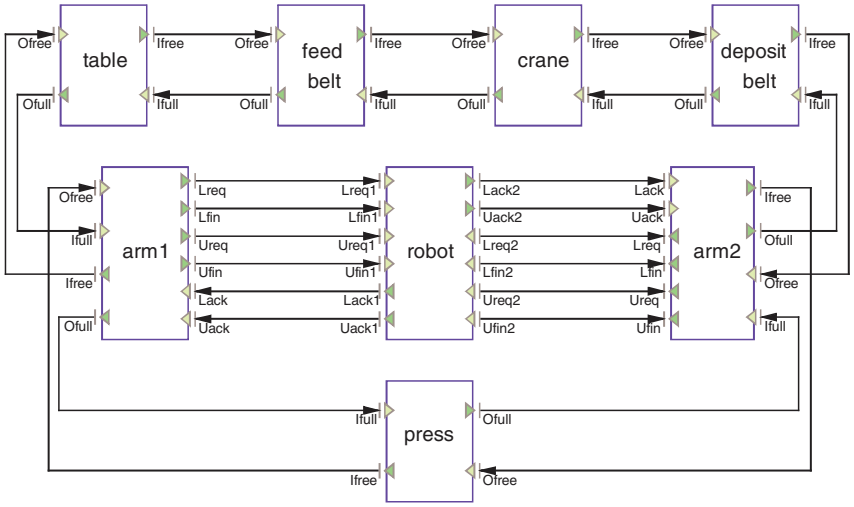



**Figure 9. Architecture of the production cell**

puts it onto the deposit belt, which transports it to the end. The system is made closed and self-contained by the addition of the crane which fetches the blank from the end of the deposit belt and returns it to the beginning of the feed belt.

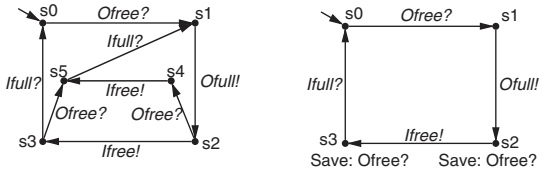
The difficulty of designing the cell lies in the fact that the cell involves 14 sensors and 13 actuators distributed over the six components. A component learns its current position by reading from its sensors and transports (or forges) the blank by controlling its actuators. The control of the actuators involves 34 commands in total [8, 9]. For example, the feed belt transports and delivers blanks to the table by starting or stopping its motor (or actuator). In order to avoid a blank being dropped outside the safe area, the belt is also equipped with a sensor detecting the arrival and departure of a blank at the end. The robot is the most complex with five groups of actuators and sensors in charge of arm extension and retraction, blank grasp, and horizontal rotation.

## 4.1. Modular design

This design is based on the Petri net model proposed by Heiner and Deussen [8, 9]. Figure 9 depicts the architecture of the design in a dataflow network. Each component in the cell corresponds to a component in the design except the robot. In [8], the robot in the cell was designed as two arm components, each of which involves a control logic for swivelling the robot to its loading or unloading angle. This is counter-intuitive to a component-based system view. Instead, we build an additional component "robot" which is in charge of robot swivel at the request of the two arms. This makes the arms identical and thus helps produce reusable implementations. The concrete robot now corresponds to three components in the design: "arm1", "arm2" and "robot". To simplify matters, we only deal with a closed system with five blanks initially residing in the feed belt, the table, the press, the deposit belt and the crane. Other variants of the system can be analysed in the same way.

### 4.1.1. Protocols

To coordinate the concurrent execution of components, we follow Heiner and Deussen's approach [8], i.e. components communicate according to the producer-consumer protocol. Figure 10(a) show the protocol specification IA for both the feed belt and the deposit belt, where the event "Ifull" (or "Ifree") locks (or unlocks) the input region for blanks and the event "Ofull" (or "Ofree") unlocks (or locks) the output region. The IA specifies the need of the feed/deposit belt for simultaneous control of input and output regions for blanks [8]. In other words, the belts cannot convey blanks unless the two regions are locked.




(a) Full version (b) Compact version

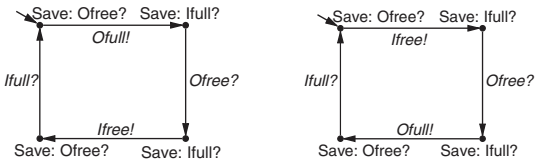
**Figure 10. feed & deposit belt IA**

Because of the reactive nature of IAs, the interleaving of input and output events often makes IA graphs messy, sometimes even for such simple protocols. To improve upon this we introduce a "save" attribute for states to specify an input event, indicating that the event is acceptable but its processing is irrelevant to the current activities. For example, the IA in figure 10(a) can be alternatively depicted



Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 IEEE

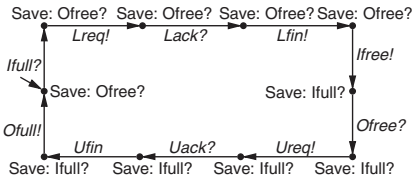
as in figure 10(b), where states which model the input of irrelevant events, e.g. $s4$ and $s5$, are removed and saved at the source states. As a result, the graph is neater and easier to understand the main factors conveyed by the protocol. Note that we only regard "save" attributes as a syntactic shortcut.



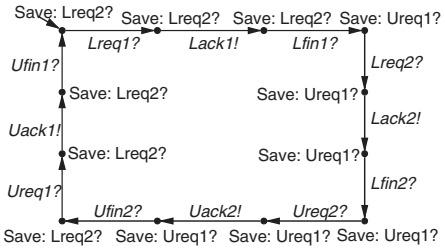

(a) table & press




(b) crane




(c) arms




(d) robot

**Figure 11. Interaction protocols**

Apart from the above, there are also two variants of the producer-consumer protocol shown in figures 11(a) and 11(b). Figure 11(a) states that the table/press must exclusively handle inputs or outputs. That is, they need to lock the input regions in order to change positions. As shown in figure 11(b), the crane needs only an independent control of its input and output regions. Put differently, it requires only one lock at a time to do its work.

To communicate with the neighbouring components, arms also follow the protocol as shown in figure 11(b). In addition, they coordinate with the robot component under a resource sharing protocol as depicted in figures 11(c) and 11(d). The protocol ensures that the sensitive operations, such as blank pickup, putdown and robot swivel, can be exclusively and atomically executed without interruption. Basically, a semaphore is exchanged between them so that only one component, the one with the semaphore, can conduct a sensitive operation. Initially, the robot owns the semaphore. An arm can ask for the semaphore by sending a request via "Lreq" or "Ureq". The request also indicates the angle at which the arm wants the robot to be, e.g. "Lreq" (or "Ureq") for the loading (or unloading) angle of the arm. Once the robot has swivelled to the requested angle, it hands over the semaphore to the arm by sending "Lack" or "Uack". When the arm finishes the sensitive operation, it returns the semaphore via "Lfin" or "Ufin". Furthermore, upon receiving a request, the robot may process it immediately or buffer it for later processing. When both arms request the semaphore at the same time, the robot will choose which one to serve. The actions taken depend on the current angle of the robot and the availability of the semaphore.

### 4.1.2. Components

With the interface automata formally describing the interaction protocol of components, we can now independently design a component with respect to an IA. Black-token Petri nets are used here as the modelling language. Although the components implement different logic for controlling actuators and sensors and collaborating with other components, they have similar structures. Here, we present the feed/deposit belt as an example.

With similar functionality and the same actuators and sensors, the feed belt and the deposit belt share a component implementation as shown in figure 12. This model is borrowed from [8, 9] but flattened and enhanced with input/output ports. Each belt contains an actuator and a sensor. The actuator is started or stopped by depositing a token into places "belt_start" or "belt_stop" (These two places are complementary.) Transitions "trans_Pstart", "trans_Cstop" and "trans_Pstop" model the process of controlling the transportation of blanks from the beginning to the end. Likewise, transitions "dlvr_Pstart", "dlvr_Cstop" and "dlvr_Pstop" model the delivery process. Whether a blank has reached or left the end of the belt is indicated by markings of the places "light_barrier_false" and "light_barrier_true" (These two should also be complementary.) The tokens in the places are only modified by the sensor as a result of occurrences of transitions "trans_csc" and "dlvr_csc".

Initially, as shown in figure 12, the belt is idle and stopped with one blank at the beginning. After it fires "lock_input" and transports the blank to the end, the belt is stopped and the light barrier is true. If the belt has received a message via "Ofree" indicating the readiness of its neighbouring component for loading, the belt proceeds to deliver the blank. This involves sequential occurrences of delivery transitions and the two unlocking transitions participating in the producer-consumer protocol. As a result, two output events "Ofull" and "Ifree" will be sequentially produced to release locks in neighbouring components.

It is worth noting that some invariants are present in



Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**

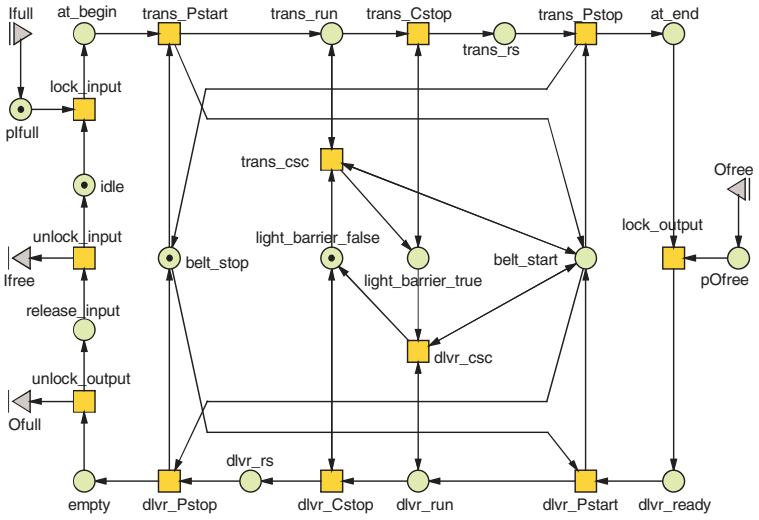



**Figure 12. Feed/deposit belt component**

this component model, e.g. the complementary places noted above. These can be ensured with purely local information, while other invariants would require information on how the environment behaves. The principle proposed in section 3 assumes the component has an environment that is as helpful as is defined by the relevant IA. This allows us to prove these invariants in the system as will be demonstrated in the next section.

In designing the concrete robot, we distribute its actuators and sensors into three components. "Arm1" and "arm2" own actuators and sensors participating in arm extension, blank grasp, and arm retraction, while "robot" manages those engaged in robot swivel. Although their models tend to be more complicated than the belt's, a similar design approach can still be applied.

## 4.2. Modular analysis

To analyse the design, we have extended our work in [14, 15] to handle mixed states in IAs, i.e. states where both input and output events originate, e.g. state "s2" in figure 10(a).

### 4.2.1. Consistency property

As stated previously, the consistency property refers to the system's freedom from deadlock and unexpected reception. To check this, we apply the modular analysis approach proposed in section 3. First of all, each component is checked for conformance with its corresponding IA. This ensures that the component does not break the guarantees specified by the IA while constructing the component local state space. This also ensures the deadlock freedom of the component in the system. We have performed these checks using the Moses tool suite and proved the conformance of every component in the design. The resultant component local state spaces turn out to be small, as shown in table 1.

| | states | transitions |
|---|---|---|
| feed/deposit belt | 21 | 28 |
| table | 44 | 66 |
| robot | 58 | 87 |
| arm1/arm2 | 60 | 90 |
| press | 36 | 54 |
| crane | 64 | 96 |

**Table 1. Component local state space**

Secondly, an abstract system is constructed which is composed of the IA in figure 10(a) and the full versions of the IAs in figure 11 with the same synchronisation structure as shown in figure 9. The consistency of this system is also ensured during the construction. The generated state space turns to be much smaller than that of the component system because we have abstracted away from the internal activity of components. Table 2 shows the size of the two systems[1]. Clearly, this approach results in significant reduc-

| system | states | transitions | construction time |
|---|---|---|---|
| Component | 1,353,857 | 5,827,108 | 45.5 min |
| IA | 692 | 1191 | $< 1$ sec |

**Table 2. System state space**

[1]The construction was executed in a SunOS 5.8 Ultra-2 machine



Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**

tion of memory usage and time consumption.

#### 4.2.2. More safety properties

In addition to the consistency property, there are also 21 safety requirements in the production cell design [8]. We classify these into three catagories: component invariants, system boundedness and system safety. Then we demonstrate how to prove them from the previously built state spaces.

**Component invariants** are predicates involving only places in a single component. Examples include the complementary places, the boundedness of places, the movement constraints on components. For example, the feed/deposit belt design requires complementary places, e.g. "belt_start" vs. "belt_stop" and "light_barrier_true" vs. "light_barrier_false" etc. Also, it is required that all places in a component can have at most one token. Furthermore, the robot is required not to rotate clockwise if its first arm points towards the table and not to rotate anticlockwise if its first arm points towards the press.

As stated previously, the invariants that hold in component local state spaces also hold in the system. To prove these invariants in the system, we only need to check the local state space of the component. We have checked all component invariants of the above-mentioned kinds in the design, including seven restrictions on component mobility given in [8]. They were found to be true in the respective component local state spaces. Hence they also hold in the system.

**System boundedness** refers to the boundedness of all places in the system. Since the boundedness of every single place can be proved as described above, we can prove that this property holds in our production cell design and the flattened net of the whole system is a 1-safe Petri net.

**System safety** refers to properties involving places in two or more components. These include 4 requirements for collision avoidance, 5 requirements inhibiting blanks being dropped outside the safe area, and 5 requirements ensuring sufficient distance between blanks [8]. We have observed that all these properties can be expressed in the following form or the conjunction of terms of this form:

$$A \implies B,$$

where $A$ involves only places in one component and $B$ involves only places in another component. For example, it is required that the feed belt conveys a blank through its light barrier only if the table is stopped in the loading position and not already loaded. This requirement can be expressed in the above form, where:

$$A = f.belt\_start \wedge (f.dlvr\_run \vee f.dlvr\_rs), \quad (1)$$
$$B = t.load\_angle \wedge t.bottom\_pos \quad (2)$$
$$\wedge\, t.stop\_horizontal \wedge t.stop\_vertical \quad (3)$$
$$\wedge\, \neg t.pIfull. \quad (4)$$

Here, $f$ and $t$ refer to the feed belt and the table, respectively. We also use place names to represent the existence of a token in the place, because the system is a 1-safe net. Line 1 indicates that a blank is being delivered. Line 2 states that the table is in the loading position. Line 3 suggests that both the actuators in charge of horizontal rotation and vertical movement of the table are stopped. Finally, line 4 implies that the table is not already loaded, where $t.pIfull$ is the successor place connected with port "Ifull" of the table and $\neg\, t.pIfull$ means the absence of tokens in the place.

To prove these properties, we follow a four-step procedure:

1. Check the local state space of the feed belt component against $A$ and calculate a set $Q$ of states where $A$ holds. The states are tuples $\langle s, q\rangle$, where $s$ a state of the feed belt IA and $q$ is a marking of the component net. Note that the IA refers to figure 10(a).

2. Obtain a subset $S$ of states of the belt IA such that $S = \{s \mid \langle s, q\rangle \in Q\}$.

3. Compute a set $S'$ of states in the table IA which are associated with a state in $S$ in the state space of the abstract system of IAs, while disregarding states of other constituent IAs.

4. We can then prove the property if $B$ holds in every state $\langle s', q'\rangle$ in the local state space of the table component for $s' \in S'$.

We have extended our previous work in [14, 15] and proved the correctness of this method. Using this method, we have successfully proved the above-mentioned 14 safety requirements.

## 5. Conclusion

We have informally presented a component-based design and analysis methodology. This uses interface automata (IAs) to capture the input-output protocol for each component, and employs a divide and conquer approach where components are tested for conformance with corresponding IAs, and the network of IAs (which matches the network of

equipped with dual 200MHZ CPU and 768MB memory. The state space of the component system is generated by the Maria tool [19] and that of the abstract system is by the Moses tool.



Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 IEEE

components) is checked for consistency. This, together with abstraction from the data values transmitted between components, leads to a significant reduction in the state space to be explored. In the case of the production cell case study which was presented in this paper, there is no abstraction of transmitted data since the system is modelled by a black-token Petri net. Still, the technique resulted in approximately three orders of magnitude improvement in the size of the state space.

We have also shown how the component-based approach can be extended to the verification of various safety properties, particularly those which can be expressed in the form (or a conjunction of terms of the form): $A \implies B$ where $A$ involves only places in one component and $B$ involves only places in another component. Essentially, the technique involves abstracting from the local component states satisfying $A$, determining the possible matching global states, projecting these global states into abstract local states for the second component, determining the matching local component states, and checking whether these states satisfy $B$. This technique was applied to the production cell case study, and was able to verify all 21 safety requirements posed in an earlier paper.

It is worth noting that, although the safety requirements studied above involve only two neighbouring components at a time, a similar approach can be taken to verify system safety properties involving multiple components. This will generalise $A$ and $B$ to be a conjunction and disjunction of component local safety properties, respectively. A detailed presentation of this work is currently being written for further publication. On the other hand, due to the over-approximation of IAs from components, it is possible for this approach to give false negatives. In this case, a process of progressive refinement will need to be adopted, as advocated in [4].

**Acknowledgements** The authors are pleased to acknowledge the helpful assistance of Monika Heiner in providing materials associated with her papers on the production cell.

## References

[1] Moses tool suite. `https://sourceforge.net/projects/mosestoolsuite`.

[2] R. Alur and T. A. Henzinger. Reactive modules. *Journal of Formal Methods in System Design*, 15(1):7–48, 1999.

[3] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the Computer Aided Verification*, LNCS 1855, pages 154–169, 2000.

[5] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Foundation of Software Engeneering*, volume 26 of *Software Engineering Notes*, pages 109–122. ACM Press, 2001.

[6] R. Esser and J. W. Janneck. Moses: A tool suite for visual modelling of discrete-event systems. In *Symposium on Visual/Multimedia Approaches to Programming and Software Engineering, HCC*, 2001.

[7] S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing*, 8(5):607–616, 1996.

[8] M. Heiner and P. Deussen. Petri net based qualitative analysis - A case study. Technical Report I-08, Brandenburg Technical University, Cottbus, 1995.

[9] M. Heiner, P. Deussen, and J. Spranger. A case study in design and verification of manufacturing system control software with hierarchical Petri nets. *Journal of Advanced Manufacturing Technology*, 15(2):139–152, 1999.

[10] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proceedings of the Computer Aided Verification*, LNCS 1427, pages 440–451, 1998.

[11] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the Computer Aided Design*, pages 245–253. IEEE Press, 2000.

[12] J. W. Janneck and R. Esser. Higher-order Petri net modeling–techniques and applications. In *Workshop on Software Engineering and Formal Methods, ICATPN*, 2002.

[13] Y. Jin, R. Esser, and J. W. Janneck. Describing the syntax and semantics of UML Statecharts in a heterogeneous modelling environment. In *Proceedings of the Diagrammatic Representation and Inference*, LNAI 2317, pages 320–334. Springer, 2002.

[14] Y. Jin, R. Esser, and C. Lakos. Lightweight consistency analysis of dataflow process networks. In *Proceedings of the Australasian Computer Science Conference*, pages 291–300, 2003.

[15] Y. Jin, R. Esser, C. Lakos, and J. W. Janneck. Modular analysis of dataflow process networks. In *Proceedings of the Fundamental Approaches to Software Engineering*, LNCS 2621, pages 184–199. Springer, 2003.

[16] C. Lewerentz and T. Lindner, editors. *Formal development of reactive systems: case study production cell*, LNCS 891. Springer-Verlag, 1995.

[17] J. Lilius and I. P. Paltor. The production cell: An exercise in formal verification of a UML model. In *Proceedings of the Hawaii International Conference on System Sciences*, 2000.

[18] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, USA, 1996.

[19] M. Mäkelä. Maria: Modular reachability analyser for algebraic system nets. In *Proceedings of the Applications and Theory of Petri Nets*, LNCS 2360, pages 434–444. Springer Verlag, 2002.

[20] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proceedings of the Computer Aided Verification*, LNCS 1427, pages 110–121, 1998.



Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)
0-7695-1949-0/03 $17.00 © 2003 **IEEE**