Modelling predictable component-based distributed control architectures

Heinz W. Schmidt, Ian D. Peake, Jue Xie, Ian Thomas
Monash University, Melbourne, Australia
{hws,ipeake,iant,jxie}@csse.monash.edu.au

Bernd J. Krämer
FernUniversität in Hagen, Germany
bernd.kraemer@fernuni-hagen.de

Alexander Fay and Peter Bort
ABB Corporate Research Center, Ladenburg, Germany
{alexander.fay,peter.bort}@de.abb.com

Abstract

Current models of component architectures require extensions to support compositional reasoning about extra-functional properties such as worst-case time. Studying such properties in architectures is complicated since actual components may not yet be chosen during architectural design, and different choices may have significant and hard to predict effects on the system properties. In this paper we show how finite state automata and Petri nets may be extended to provide compositionality of extra-functional properties. We focus on worst-case time and safety. We illustrate the use of these techniques on the well-known production cell case study. In collaboration with industry we are currently applying a prototype analysis system for predicting critical properties of real-time industrial control systems.

1. Introduction

The software architect is concerned with both functional and extra-functional design. In component-based architectures, the former task can involve specifying, choosing, adapting, analysing and connecting components. The latter task involves analysis over properties such as reliability, availability, or performance. As system complexity increases so do the costs associated with those tasks, even more so if heterogeneous methods are used. It is therefore desirable to define methods and formalisms that can simultaneously address functional and extra-functional concerns of component-based architectures.

Components provide interfaces for their environment of use but also interfaces specifying services required of other components. Our models of architectures and components are *parameterised* to support component adaptations during architecture design. Component parameterisation is a general concept: (1) a partial interface is identified as the formal parameter; (2) the parameterised interface acquires a semantics by associating the formal parameter with some constraint or bound representing the class of possible parameter actualisations; (3) formal parameters are replaced by representatives of this class to actualise components. Syntactic parameters decorate signatures, for example, by associating transition names with transition probabilities. Structural parameters include required interfaces or subcomponents replaceable only under certain constraints such as conformance. Semantic parameters include changing constraints and interpretations expressed, for example, by recursive function definitions or inductively defined equality. In [15] we utilised language inclusion to permit syntactic and structural parameterisation.

Parameterised contracts were initially developed for sequential systems using Finite State Automata (FSA) for protocol specifications [11]. This work was later extended towards parameterisation for concurrent and open distributed systems and led to semi-automatic adaptation of parameterised concurrent protocols [14]. In this paper we illustrate the advantages of FSA-decomposable Petri nets in facilitating compositional reasoning about both functional and extra-functional properties of distributed control systems. We focus on safety properties and worst-case execution time and show the use of Petri nets in combination with parameterised components and architectures. The well-known production cell example [6] serves to illustrate our algorithms. They are currently implemented in the context of a commercial engineering design environment.

The paper is organised as follows. The production cell is sketched in Section 2. Section 3 introduces Petri nets and FSAs and shows the derivation of a formal design model of the production cell's control software. In Section 4 we study safety issues and worst-case execution times. Section 5 relates our modelling approach to a programming standard for concurrent control applications used in industry. Section 6 discusses related work.

Proceedings of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03)
0-7695-2054-5/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

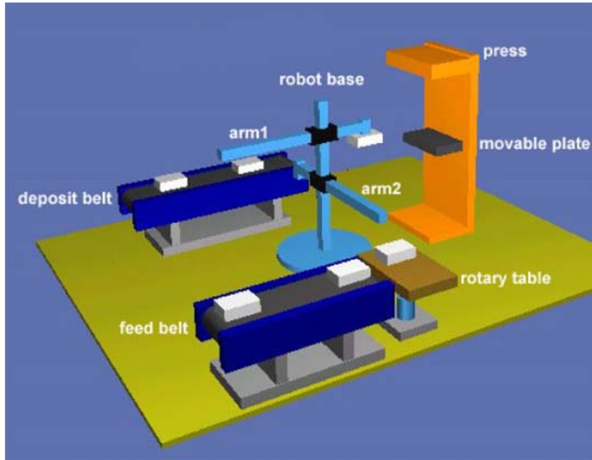

Figure 1. Production Cell

2. Example Application

To illustrate our concepts and models we use a variant of the production cell introduced in [6] and explored in [5]. In this example we assume an external gadget to deposit individual metal blanks on the left end of the feed belt (left front) one after the other in arbitrary time intervals. The belt conveys these blanks to the rotary table at the belt's other end. Once the blank has been passed over, the table moves up and slightly rotates to bring the blank into a position and onto a level from where robot arm 1 can pick it up. After arm 1 has been loaded, the robot rotates counter-clockwise into a position in which arm 1 points at the press to introduce the blank into the press where it is forged while the movable plate of the press is closed. Once the last blank is processed or when for other reasons there is no blank at the time arm 1 is ready to load a blank, the cell must continue processing the metal pieces in progress. Sensors detect the empty rotary table in this case and the press will skip its forging action by holding the plate in its current position.

To increase the use of the press, the robot is equipped with a second arm. Arm 2 picks up the part forged in the previous cycle from the press, the robot rotates counter-clockwise until arm 2 points toward the deposit belt and unloads the piece of metal on that belt. This belt moves processed parts to its other end. From here they are removed one by one by a second gadget in the environment. Arms 1 and 2 are mounted on different vertical levels and access the press while its movable plate stays at two different vertical levels (middle and bottom, respectively). The arms can carry load with their magnetic grippers. The arms can individually be extended or retracted horizontally but must always rotate together as they are strongly coupled with the rotating robot base. Again, sensors monitor the load and unload zones of the deposit belts and signal the actual actuator positions.

We assume that the robot, press and the conveyor belts have their own software controller, whose behaviour we will formally model. These controllers communicate by some form of message or signal passing.

The critical sequence of robot actions is as follows: first, arm 1 takes a blank from the rotary table; then in successive left rotations, arm 2 takes the last forged piece from the press, arm 1 unloads the blank on the empty press plate and arm 2 unloads the forged piece on the deposit belt. Finally, the robot moves arm 1 back to the rotary table and this sequence starts over again. For simplicity we assume an abstract view of the interaction between the controller and the components, which hides details about synchronisation, sensors, or fine motor control. We enumerate the abstract operations in Table 1 along with the worst-case execution time for the corresponding elementary physical operations.

`take` (`drop`)	take (drop) load with gripper magnet	10
`ldown` (`lup`)	lower (raise) press plate	50
`ext` (`ret`)	extend (retract) arm	300
`tl` (`tr`)	rotate robot base by one position counterclockwise (clockwise)	400
`press`	press blank	200
`hold`	do not press—no blank present	200

Table 1. Cell actions with worst-case time

3. Behaviours and Composition

3.1. Traces

Although FSAs are traditionally well suited for modelling discrete sequential system behaviours, Petri nets provide more succinct presentations of physical systems and mature abstractions for true concurrency. For analysing extra-functional properties in real-time distributed software systems, we wish to combine the strengths of FSA methods with those of Petri nets.

In the project presented here we have focussed on *rational trace languages*. They have been studied from the viewpoint of automata and Petri nets with the aim to characterise traces in parallel systems. Traces are modelled by partial orders with repeated occurrences of symbols from an alphabet Σ. Traces are further equipped with a finite reflexive symmetric *dependency* relation D requiring that dependent and repeated symbols must be ordered. Thus independence (the complement of dependence) models parallelism. Each trace can also be viewed as an equivalence class of all possible linearisations of the trace. There is a correspondence

between regular languages for such linearisations and rational trace languages. Correspondingly a certain class of Petri nets characterises rational trace languages. The reachability graph of a Petri net is the FSA accepting these regular linearisations. In this FSA true parallelism has been 'lost'. The dependency relation and its complement, independence, can be computed from the net, however, and associated with the FSA alphabet to 'record' true parallelism and permit the reconstruction of a behaviourally equivalent Petri net from its FSA paired with an independence relation [9].

The practical use of FSAs for parallel systems is limited by the combinatorial explosion in the number of states needed to model complex distributed components. This becomes apparent in the parallel unsynchronised composition of two FSAs as their shuffle product that contains all possible permutations of pair wise independent sets of transitions and all possible intermediate states for any such permutation. In contrast, the Petri net representation of this product is the graph union of the two nets (placing the two nets side by side and possibly merging identical transitions). The size of this 'net product' is linear in the size of the component nets. Even with synchronisation the composite net preserves the original component at least syntactically.

Rational trace languages, and the Petri nets that generate them, are closed under rational operators including concatenation (;), Kleene star (*), intersection (&), union (|) and parallel product (||). They are recognisable if they are star-connected. Star-connectivity loosely relates to a weak fairness under projections to sequential components. [9] has shown that this is a necessary and sufficient condition for characterising these trace languages.

3.2. FSA-decomposable Petri nets

In our work we exploit the equivalence of FSAs over independence alphabets with a corresponding class of Petri nets built from sequential components using regular operators and preserving star-connectivity. For simplicity, we call these nets *FSA-decomposable Petri nets*.

We illustrate net construction and notation via a running example, constructing a net modelling the operation of the production cell. Fig. 2 shows (i) a simple `Switch` to the left followed by structurally equivalent nets (ii) for a `Expander`, (iii) a `Gripper`, and (iv) an unconstrained robot base controler. Tokens (black dots) mark the initial *places* (or states) in a net. Boxes represent actions. Performing the action requires tokens on all states in its *preset*, removes them from the preset and places them on its *postset*.

Each of these nets is sequential and thus identical to its FSA, with the initial state being marked and the independence relation being empty. These sequential nets, also called *S-components*, are atomic from the viewpoint of parallel composition. S-components have exactly one token in any state. Flow in S-components branches in states (modelling choice) but not in transitions (modelling synchronisation). Their concatenation (merging the final state of one with the initial state of the other) leads to an S-component.

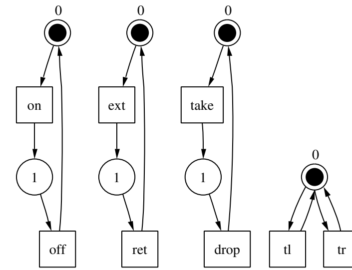

Figure 2. FSA component nets, left-to-right: (i) `Switch`, (ii) `Expander`, (iii) `Gripper` (iv) `UnconstrainedBase`

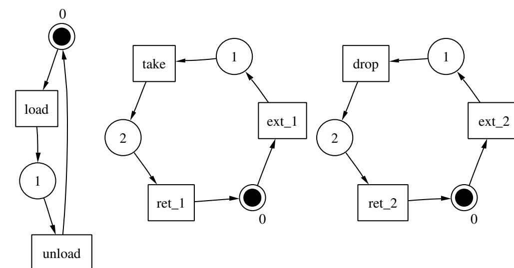

Figure 3. FSA component nets, (v) `AbstArm`, (vi) `load` and (vii) `unload`

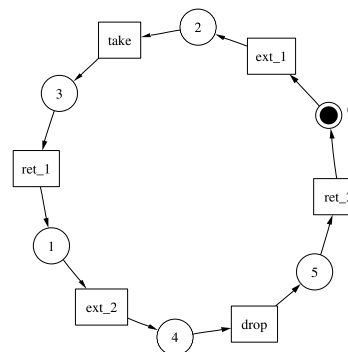

Figure 4. `RefinedArm` as the substitution of `load` and `unload` in `AbstArm`.

IEEE COMPUTER SOCIETY

Their union (which results from merging two S-components with disjoint transitions but possibly common states) leads to an S-component. Their iteration leads to a cyclic structure with identical initial and final state suitable to model *periodic* real-time systems.

The nets (ii, iii, v) in Figs. 2 and 3 are derived by renaming symbols in the action alphabet. Fig. 3 shows an *action refinement*: `load` and `unload` nets refine the corresponding actions in `AbstArm`. By convention, the refinement net is a single cyclic run from remarking the initial state (0). The refinement resulting from substituting these nets for the corresponding actions is depicted in Fig. 4.

`Arm=Expander||Gripper` denotes the *free parallel product* of two nets. It is visualised by viewing only nets (ii) and (iii) together (as the union of these two graphs). The reachability graph is the shuffle product of the corresponding FSAs. Net products are visualised as unions, in which states are disjoint and actions may be shared. Shared actions lead to mutual restrictions. Thus the product `Arm=Expander||Gripper||RefinedArm` combines the components `Expander` and `Gripper` under the constraint `RefinedArm`; the constraint results from sharing actions. The resulting net is behaviourally equivalent to that in Fig. 4 but contains additional dependencies. For example, a copy of the `Gripper` state 1 would appear between `take` and `drop` in the unreduced product.

Using these composition operations, nets are built hierarchically to model component interfaces and behaviours. Components are assumed to be either primitive, i.e., predefined and selected from a library or converted from an FSA, or derived by recursive composition and transformations. Refinements correspond to substitutions in the net.

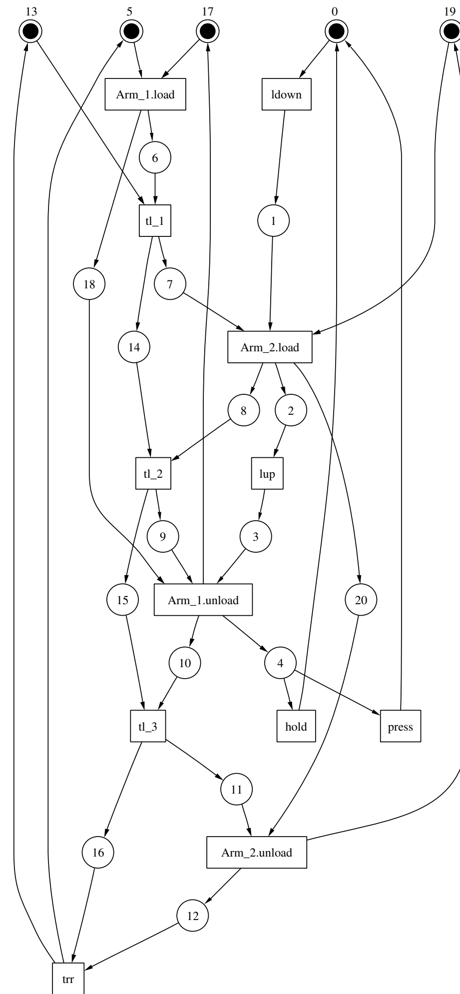

Figure 5. Production cell core combining robot and press

4. Extra-Functional requirements

Fig. 5 shows the Petri net of the core cell as the composition of the `Press` and `Robot`. The latter is a composite as well built from subcomponents `Arm1`, `Arm2` and `Base`, with `tl` for turn left, `tr` for turn right and `trr` for turn right return, consisting of three successive right turns of the robot base rotation unit.

4.1. Synchronisation

For visualising synchronisation and reasoning about it, the net representation can be more convenient than the FSA representation. Due to the correspondence of FSA-decomposable Petri Nets and FSAs over independence alphabets, we can switch representations without losing compositionality. We can also take advantage of algorithms developed in either of the two, as appropriate.

Because synchronisation constraints share actions with the components they synchronise in net products, they introduce additional dependencies and orderings. For example, the order of `Expander` and `Gripper` actions within the `Arm1` component enforces that the gripper magnet operates exclusively while the arm is extended (see Fig. 4). This constraint will be observed in subsequent compositions such as shown in Fig. 5.

Fig. 6 (viii) shows the robot base rotation synchronised with the movement of the arms. The second constraint (ix) in that figure controls the interaction of the press with the robot components `Arm_1` and `Arm_2`, which are instances of the `AbstArm` component type shown in Fig. 4.

We distinguish *hidden* and *shared* actions. Only the lat-

0-7695-2054-5/04 $20.00 © 2004 **IEEE**
IEEE COMPUTER SOCIETY

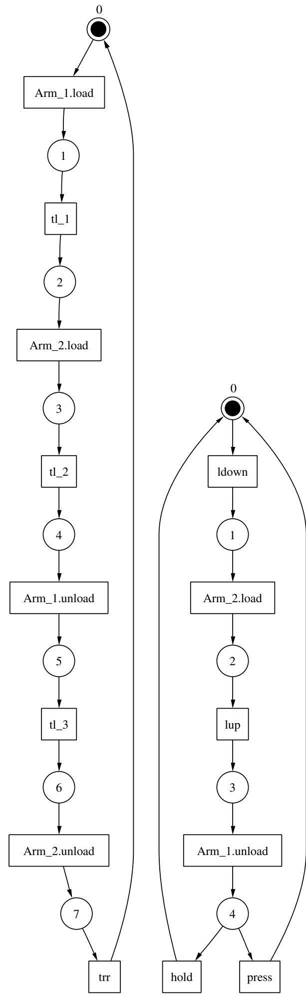

Figure 6. Synchronisation constraints for: (viii) robot base and arm movements (left) and (ix) robot-press interaction (right)

ter may be shared by parallel components, and are therefore *controllable* by other components via synchronisation constraints. Hidden actions represent actions not controllable externally.

In Petri nets we regard transitions as instantaneous and timeless, while duration is associated with states. Actions like a robot rotation `tl` have non-negligible duration. They are associated with a clear instant beginning `tl!beg`, and end `tl!end` and a state of execution `tl!do`. In our nets we assume all actions have such an implicit refinement.

4.2. Liveness

A strong form of liveness is the possibility of every transition to reoccur eventually whatever state is reached. This is particularly important for periodic systems. Since our systems are FSA-decomposable by construction, an obvious candidate for liveness and safety analysis are algorithms based on Commoner and Holt's theorems [1], which require nets to be covered by marked S-components. Our nets are partitioned in this sense by construction. For periodic system models they also return to the initial state and often exhibit a *local choice* property, where conflicts (alternative actions with shared preconditions) are not shared between parallel components. For such systems the only other property to check is that the parallel product does not accidentally introduce unmarked S-components via shared actions. Such S-components indicate local deadlocks as no transition in it will ever fire and hence the net is not live. The above core cell abstraction can be proven live by confirming that all cycles run through initial markings, no unmarked cycle exists as a result of parallel composition and hence no unmarked periodic S-component exists.

4.3. Parameterised Behaviours

Each component in our model is associated with an abstraction consisting of three parts: its provided protocol, behaviour abstraction and its required protocol. Each of these are represented by nets. For each extra-functional property we enrich our models with appropriate information. For example, reliability is associated to states and transition probabilities to transitions.

In our system we package the relevant nets and property decorations with the component descriptions into *property-enriched components*. The compositionality of FSA-decomposable nets permits us to analyse such models with generic parameters and store relevant component properties resulting from such analyses when these parameterised components are filed in a library. Thereby we avoid re-computation of certain properties when components are reused in new designs.

When modelling extra-functional properties such as safety or reliability, using parameterised models becomes a necessity because component-based subsystems are intrinsically *open*. Inevitably some of the information required to predict accurately a relevant component property will be available only in the context of deployment. A parameterised model can be actualised by the relevant parameter. We use composition expression with component variables. For example, a composite behaviour may be parameterised by different synchronisation constraints for the core cell. The actual parameter may only be chosen at a later design stage, at configuration or deployment time.

IEEE COMPUTER SOCIETY

4.4. Safety

To demonstrate the use of our nets to facilitate the expression and proof of safety properties, we enumerate informally a selection of the safety requirements of the overall system, then show how such requirements can be expressed formally and proven. The safety requirements R1-R3 are taken from [6]: R1 restricts machine mobility. R2 and R3 avoid machine collisions.

R1: The robot must not be rotated clockwise in its rightmost position, where it picks up blanks, nor counterclockwise in its leftmost position, where it deposits pressed metals.

R2: The press may only close when no robot arm is positioned inside it.

R3: A robot arm may only rotate in the proximity of the press if the arm is retracted or if the press is in its upper or lower position.

We can formalise safety properties by associating assertions with states in nets. A marking of the net is then interpreted as the conjunction of corresponding assertions. The net can be viewed as a specification of a logical system. The connection between linear temporal logic and Petri nets is well-established [2]. An LTL prover can be used to statically verify aspects of the specification.

We analyse a net by simulation (according to the well-known firing rules), which amounts to model-checking by iterating the net reproducing the initial state. At each state we can check the resulting assertion for violations of specified safety constraints.

Assertions may include arithmetic expressions of the form $\#t$, which give us the number of executions of transition t in some net. We limit counter arithmetic to linear operators and Boolean comparison. In general, safety properties can be expressed as invariants over protocol states. Some such invariants are local to components. For example, the safety requirement R1 can be formalised as follows:

$$(\mathrm{F1})\ 0 \leq \#\mathtt{tl} - \#\mathtt{tr} \leq 3$$

The unconstrained robot base (shown in Fig. 2(iv)) permits traces in which F1 is violated. The constraint realised in Fig. 6(viii) controls these actions and satisfies F1. Because (viii) is an S-component in the core cell, then the core cell satisfies F1, by inspection.

The validity of assertions can be lifted from states to traces. The last actions in a (partially ordered) trace lead to a unique set of states immediately enabled by these actions. Thus a trace leads to the validity of a formula via its immediately following states and the assertions they satisfy. This then becomes the basis for inductive proofs on traces. For example, the initial state of (viii) satisfies F1 trivially (counters are 0 initially). Assuming F1 is satisfied for traces of size n (number of actions), we can now prove case-by-case that firing the next action preserves F1.

Although our tool does not yet support inductive proofs, the net representation lends itself to such proofs of safety properties. Given a safety formula ϕ as a set of (conjunctions of) existing net conditions; in the base case we have to establish that ϕ is a subset of the initial state; then iterating over all transitions, for the induction step, we show that the current transition inspected either cannot be enabled under ϕ or if it can, its firing will not result in a violation of ϕ.

4.5. Worst-case execution time

In [13] we have shown how metrics on execution traces can be used as a basis for a calculus on space and time properties. In an industrial collaboration with ABB in the area of component-based distributed control systems, we are now developing component-based methods for predicting worst-case time based on parameterised FSAs and Petri nets with time specification. We use hierarchical FSAs and Petri nets preserving the composition hierarchy and various parameterisation mechanisms to permit a one pass algorithm for the worst-case execution time prediction. We assume bounds for loops and rely on a well-formedness restriction for intertwined loops.

On our running example of the production cell the analysis degenerates to the special case of determining the longest weighted cycle (by worst-case time) in the net of Fig. 5. The longest weighted cycle in this net is in fact the robot arm and base synchronisation net shown in Fig. 6 (viii). The weighted length of this cycle is $4*610+6*400 = 4840\ (4.84s)$.

Restrictions on products may effect the time by ordering unordered events. When a component is reused in such a product, it may necessitate changes in the equations representing the time model, and hence re-computation of the times. Such re-computation may also become necessary if some of the external action times are variables. Since this is a single pass algorithm over the states and transitions of the protocol type, it is, however, realistic to postpone computation of the parameterised time to the point of reuse, reconfiguration or deployment. With these caveats our models are compositional as they are strictly separate from the implementation and generally require much less detail.

In general, worst-case execution time models are only compositional under certain sufficient constraints for well-behaviour. For example, if we rely on failure-free synchronisation and deadlock freedom, our time models are compositional. This reliance is a proof obligation that may require access to the behavioural abstractions of the components.

The running example is a model for the robot cell in its physical environment. Computing the worst-case times in this model relates to estimates for the physical time of motor movements, sensor and actuator processing.

IEEE COMPUTER SOCIETY

5. Design for prediction and verification

Compositional approaches to design are used widely in engineering. The authors from Monash University are collaborating with ABB's industrial automation research group to enable the prediction of extra functional properties such as time, space and reliability in ABB's engineering development platform.

We have developed customised versions of our models for worst-case time analysis and prediction (TCAP) of designs according to the IEC 61131-3 standard, a design and programming notation for distributed control systems. In IEC 61131-3 function block notation, components can be viewed as concurrent entities synchronised via their interfaces. Our approach to modelling function block designs exploits this view by applying component-based modelling techniques as discussed above.

5.1. The ABB Aspect Integration

We are currently realising TCAP as a new aspect within ABB's Aspect Integration Platform, which already contains editing and design tools for function blocks.

This platform particularly addresses integration problems of different steps in engineering industrial control systems such as the selection of the control system hardware structure, the implementation and test of the controller software, or the installation and commissioning of the control system. Although these steps are already supported by state-of-the-art methods and comprehensive software tools, the results of the individual engineering steps remain isolated because the tools are not integrated, in general. Consistency between concurrent engineering tasks and along a plant's life cycle is therefore not yet achieved.

Several means are being developed at ABB to address this issue. One solution, called "Aspect Objects", is based on the notion of all information being ordered and assigned to certain software objects. They represent objects of the real plant, such as motors, valves, or other physical entities. Each aspect maintains all relevant information that is of interest to a certain group of system developers or users. For example, a motor has a "mechanical dimension aspect" comprising mechanical parameters of interest to motor designers and persons who mount or dismount the motor in the plant. A motor may also possess a "control aspect", which comprises the control code to start, run, and stop the motor and which is of interest for the control engineer who designs the control concept and for the operator in case of motor starter problems. The information in each aspect is managed by a particular tool. Thus, the concept of Aspect Objects allows us to couple different tools via a common object.

From a control engineering point of view, the control aspect is the most relevant aspect to consider since it holds for each relevant device (at least for sensors and actuators) the corresponding control code to operate (i.e., parameterise, start, run, stop, and maintain) this device. Furthermore, it contains control code that interconnects devices, such as continuous control loops and binary cause-effect interlockings.

5.2 IEC 61131-3 Function Designer

The design and implementation of the control code can be carried out by a variety of programming or engineering tools. The selection of the tool can be based on criteria such as support of the standard IEC 61131-3 programming languages, controller hardware supported by the tool, intuitive and easy use of the tool, or support for efficient engineering of very large projects.

An engineering tool with special strengths regarding the latter criterion (e.g. by providing extensive cross reference lists to track parameter and variable usage) is the "Function Designer". Among others, it allows the programming of large control code networks by instantiating, parameterising, and connecting function blocks according to IEC 61131-3. It will be closely interacting with novel tools under construction implementing the analysis methods presented in this paper.

6. Related and Future Work

Our approach to syntax and overall behavioural semantics is similar to that of in some architecture description languages [8] [4] [7]. However these approaches do not model true concurrency and they lack support of extra-functional properties.

In a series of papers we have explored our approach for a range of extra-functional areas, such as transactional contexts [10], adaptability [15], reliability [12] and fault-tolerance [3].

A variant of the production cell has been used in [16] in combination with the authors' preferred modelling technique, coordinated actions, to illustrate the technique's capabilities to formalise and examine fault-tolerance and safety properties.

The focus of the work presented here is somewhat different. We want to enrich architecture and component models by extra-functional properties, notably real time, with the aim of making such specifications accessible to formal analysis and at the same time scalable, i.e., compositional, in a distributed real-time application domain.

While FSA and Petri net models are well-known notations for protocol specification and adaptor generation [17],

our use of parameterisation and its combination with architectural abstraction appears novel.

7 Conclusions

The functionality provided by a component always depends on the functionality received from its environment. Hence, especially in those systems where many different configurations and environments exist, the ability to perform fine-grained adaptations and to model and predict extra-functional properties is crucial.

In our work we adapted FSA and Petri net models to cater for compositionality and extra-functional properties. These models provide an adequate basis for prediction and verification of liveness and safety properties including collision freedom, deadlock freedom and others. For safe systems a rich set of analysis is then available in the theory of Petri nets to guarantee liveness and fairness properties [1].

We have shown in previous and the presented work how these can be adapted and utilised in practical development environments such as the presented AIP-TCAP aspect.

Finally we have sketched an algorithm for compositional worst-case time prediction and analysis based on hierarchical Petri nets, composed from sequential components.

These nets and the underlying FSA models are used a basis for interpreting IEC 61131-3 designs and efficiently predicting worst-case execution time, provided that these nets pass the liveness and safeness analysis.

The implementation in an industry-strength engineering environment for industrial automation was sketched.

References

[1] E. Best and P.S. Thiagarajan. Some classes of live and safe petri nets. In *Concurrency and Nets*, LNCS, 72-94, Springer, 1987.

[2] U.H. Engberg and G. Winskel. Linear logic on Petri nets. Technical Report BRICS RS-94-3, University Aarhus, Computer Science, February 1994.

[3] J. Jayaputera, I.H. Poernomo, and H.W. Schmidt. Timed probabilistic reasoning on UML specialization for fault tolerant component based architectures. In *SAVCBS 2003 Proc., Specification and Verificatin of Component-Based Systems*, 42–48, 2003.

[4] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, 2000.

[5] B.J. Krämer. A case study in developing complex safety critical systems. In *HICSS*, Vol. V: Advanced Technology, 135–143, 1997.

[6] C. Lewerentz and T. Lindner (Eds.). *Formal Development of Reactive Systems - Case Study Production Cell*. LNCS 891, Springer, 1995.

[7] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley & Sons, 1999.

[8] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[9] E. Ochmanski. Recognizable trace languages. In Volker Diekert and Grzegorz Rosenberg, editors, *The Book of Traces*, pages 165–203. World Scientific, 1995.

[10] I.H. Poernomo, R.H. Reussner, and H.W. Schmidt. Architectures of Enterprise Systems: Modelling Transactional Contexts. In *Proc. of the First IFIP/ACM Working Conference on Component Deployment (CD 2002)*, LNCS 2370, 233-243, Springer, 2002.

[11] R.H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *34th HICSS*, IEEE, January 3–5 2001.

[12] R.H. Reussner, H.W. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66(3):241–252, 2003.

[13] H.W. Schmidt and W. Zimmermann. A complexity calculus for object-oriented programs. *Journal of Object-Oriented Systems*, 1(2):117–147, 1994.

[14] H.W. Schmidt. Trustworthy components: Compositionality and prediction. *Journal of Systems and Software*, 65(3):215–225, 2003.

[15] H.W. Schmidt and R.H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronisation. In *Proc. of the Fifth IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, 213–229. Kluwer Academic Publisher, March 2002.

[16] J. Xu, B. Randel, A. Romanovsky, R.J. Stroud, A.V. Zorzo, E. Canver, and F. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Transactions on Computers*, 51(2):164–179, 2002.

[17] D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.

0-7695-2054-5/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY