Towards Design Verification and Validation at Multiple Levels of Abstraction*

Correct Design of Distributed Production Control Systems

Holger Giese, Martin Kardos, and Ulrich Nickel
University of Paderborn, Germany

Abstract: The specification of software for distributed production control systems is an error prone task. The ISILEIT project aims at the development of a seamless methodology for the integrated design, analysis and validation of such embedded systems. Suitable subsets of UML and SDL for the design of such systems are therefore identified in a first step. The paper then focuses on how we use a series of formal semantics of our design language to enable the effective evaluation of software designs by means of validation and verification. We will further explain how the use of multiple Abstract State Machine meta-models permits simulation and model checking at different levels of abstraction

Key words: embedded system design, UML, SDL, formal semantics meta-models, ASM, formal verification, model-checking, validation, code generation

1. INTRODUCTION

In today's embedded systems, the fraction of hardware, which realizes the functionality of the system is decreasing and replaced by decentralized and complex software systems. Modeling approaches and software development processes and techniques for the production of correct, stable and flexible adjustable distributed embedded software systems are thus required. Integrating software development into the overall system engineering process is one step. Moreover, analysis techniques, that fulfill the additional requirements system engineering brings into the game, have to be embedded.

* This work has been supported by the German Research Foundation (SPP1064, GA 456/7).

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35599-3_29

B. Kleinjohann et al. (eds.), *Design and Analysis of Distributed Embedded Systems*
© IFIP International Federation for Information Processing 2002

The ISILEIT project [1] aims at the development of a seamless methodology for the integrated design, analysis and validation of distributed embedded production control systems. Its focus is the (re-)use of accepted techniques which should be improved with respect to formal analysis, simulation and automatic code generation.

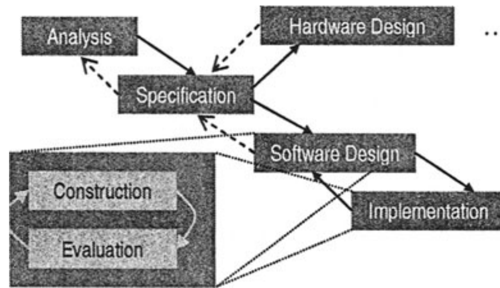

Figure 1. Construction and evaluation during design

The methodology defined in the ISILEIT project consists of several consecutive design steps. Together, they form a consistent development process that covers all aspects of system design, i.e. analysis, specification, design and implementation (see partially Figure 1). At the beginning of every system design process, a core system specification has to be developed. This specification then further serves as an input to the design process for the different system elements including the software. In the overall process, effects backwards are somehow required to be minimal (dashed arrows), because changes in hardware would result in high costs. In contrast, in the later phases of the software design and implementation, we have a more iterative process where construction and evaluation alternates to ensure that the specification is met (see Figure 1).

To describe embedded system software during the *construction* phase with UML [2] and SDL [3], we integrated SDL block diagrams, UML statecharts and collaboration diagrams [4] to form an executable specification language that allows us to specify reactive behaviour as well as complex application specific object structures.

The *evaluation* of a particular possible incomplete software model against the requirements imposed on its behaviour is also part of the design. A number of informal techniques, such as reviews or walkthroughs can be used. However, informal approaches often fail to identify the misconceptions related to the complex interplay of multiple processes and timing effects present in distributed embedded systems. Therefore, in our approach also the formal system evaluation is supported in form of (1) verification by means of model-checking and (2) validation by means of

simulation or testing. We adapted Abstract State Machine (ASM) [5] as a formalism that helps us to tailor the system models to any given or desired level of abstraction by means of different meta-models. This allows us to manipulate the designed system state-space and its size. The *validation* in our approach is based on this series of formal semantics with different levels of abstraction or running the generated executable code. For the automatic *formal verification* of a designed system model, a transformation is used, that generates a corresponding low-level model-checking description based on its ASM model and it's meta-model. Such a low-level model description can be model-checked and the results are propagated back to the design view.

In Section 2 the design language and modelling activities during the construction are described. The support for multiple levels of abstraction and the verification are then considered in Section 3. In Section 4 the validation of a design at different abstraction levels and the implementation model are presented. The paper closes with an overview about related work in Section 5 and a conclusion and outlook on future work (Section 6).

2. CONSTRUCTION

The major emphasis of the ISILEIT project lies on (re-)using existing techniques, which are used by engineers in industry for the specification of the control software of a material flow system (MFS). Thus, during the first phase of the project, we analysed our case study to identify which specification technique is most suitable for which part of the system specification with respect to the domain of production control systems. Figure 2 shows the schematic overview of the regarded system.

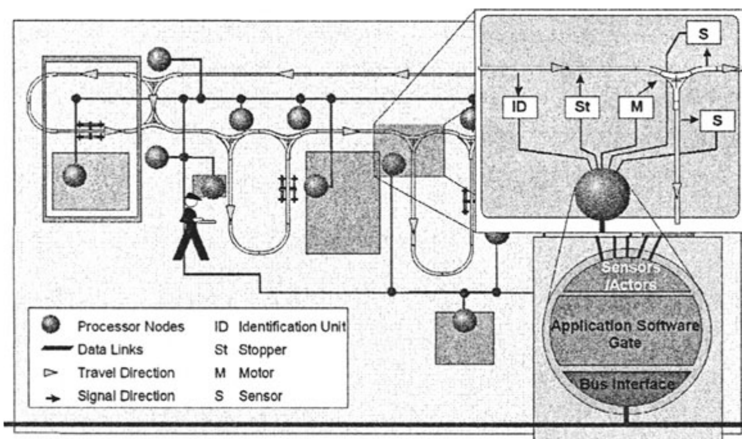

Figure 2. Schematic overview of the material flow system (MFS)

The material flow system, which connects several manual working stations and robots, is track based. The MFS is controlled by processor nodes, which are connected by a field bus. Every node is responsible for a part of this distributed system. In Figure 2, a processor node is depicted, which controls a gate. The sensors and actors are connected with the node by the process interface.

The here described system can be seen as a system of asynchronously communicating automata. Therefore, we decided to take SDL block diagrams to specify the overall static communication structure. SDL is a design language, which is well known from the area of telecommunication engineering. It has a well defined formal semantics, which captures the asynchronous communication aspects of our system [6]. We integrated parts of the available ASM specification of SDL into the formal semantics of our design language. Figure 3 shows the result of the first step. It is a simple SDL block diagram showing the system as a network of logical control nodes – processes running in parallel and communicating via signal channels.

Figure 3. SDL block diagram of the sample factory

Figure 4. Simple statechart describing the shuttle control

We derive the initial (UML) class diagram of the desired system. At this, each process(type) identified in the SDL block diagram generates a class in the class diagram. In addition, each signal received by a process in an SDL block diagram creates a signal method in the corresponding class. In the next step, we can now refine this class diagram by adding further classes, which the active classes use to implement their functionality.

SDL block diagrams (and the derived class diagrams) specify the number of signals which are understood by the different processes. Now, we have to define how each process will react on these signals. Thus, for each process

class, one has to provide a statechart modelling the general process behaviour. These statecharts should at least cover all signals that are understood by/declared in the corresponding process class. In the engineering field, usually SDL process diagrams are used for this purpose. However, we prefer statecharts here, due to the additional expressive power of nested states and history states. Following this idea, we assign to every active class (process) one UML statechart that describes its behaviour. Figure 4 shows the statechart that specifies the behaviour of the process (or active class) shuttle.

In current practice, when the detailed behaviour of the system needs to be specified, pseudo-code or statements of the target language are used. To avoid this in our approach, we decided to use a well defined subset of activity and collaboration diagrams which are combined to so called story diagrams. We use this visual programming language for the specification of do, entry, exit, and transition actions within the statecharts.

To conclude, we integrate SDL block diagrams, statecharts and collaboration diagrams to form an executable design language that allows us to specify reactive behaviour as well as complex application specific object-structures. A more detailed description of the presented methodology can be found in [4].

3. VERIFICATION

ASM serves in our approach as method that integrates all modelling languages combined in our graphical modelling language at one common semantic base. In other words, it models the semantics of this language. In our approach, we model distributed control systems with possibly complex behaviours that imply large system state spaces. Therefore, different levels of abstraction can be achieved using multiple ASM meta-models and only a single ASM encoding of the concrete system model (see Figure 5). The ASM meta-models are operating as interpreters of the specific ASM encoding of the concrete system model and automatically abstract from not required details.

The verification of a designed system model is based on the generation of an ASM encoding of the system model. This is done by filling in the unspecified data structure (domains and functions) of an ASM meta-model. For example, the sets of states, transitions and events of the statechart from Figure 4 form the data structure that will be filled in ASM meta-model I. The required complete ASM model is derived by combining an ASM meta-model with the specific ASM encoding of a concrete system. This complete ASM model is further fed into a transformation process that generates a corresponding low-level model-checking description, e.g., a description in a

specific model-checker language [7] or a particular BDD representation. Finally, this low-level model description can be model-checked and the results can be propagated back to the system design.

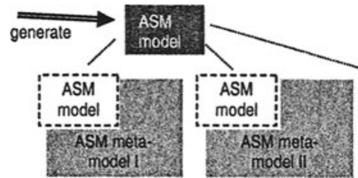

Figure 5. Constructing the ASM model

In the most abstract view (meta-model I), we consider the processes just in a black box view. We abstract from the concrete communication behavior assuming only the non deterministic occurrence of events that are accepted by the process, i.e. the rule *selectEvent* in Figure 6 left-hand-side non-deterministically choose one element from the set of all events acceptable by the process (cf. pessimistic abstraction [8]). Also the state representation together with actions and guards are ignored in order to hide any implementation details. Therefore, the guard evaluation rule *evalGuard* chooses non-deterministically between true or false and action evaluation rule *executeAction* executes just an empty rule *skip*. Thus, the result is an ASM meta-model describing the semantics of a statechart ignoring all implementation details related to events, guards and actions.

I

```
// evaluate specified guard
evalGuard(g as Guard) as Boolean
= choose v in {true, false}

// select event for dispatching
// chose any possible one
selectEvent() as Event =
choose e in acceptable_events

// execute specified action
executeAction(a as Action) = skip
```

II

```
// evaluate specified guard
evalGuard(g as Guard) as Boolean
= choose v in {true, false}

// select event for dispatching
// take first event (FIFO queue)
selectEvent(a as Agent) as Event =
a.events:= tail(a.events)
return a.events(0)

// execute specified action
executeAction(a as Action) = skip
```

Figure 6. Sample of abstractions in ASM meta-model I and II

It is obvious that this level is sometimes too abstract and restricts the set of verifiable properties to a small subset working just without any bounds to implementation (e.g., the object hierarchy), but it serves as a good base for

the further refinement going down from this very high-level ASM meta-model to more detailed ones. This abstract view can be step by step refined taking into account more and more details of the implementation: variables, the object structure etc.

In Figure 6 right-hand-side the *selectEvent* rule of the refined meta-model II is presented. In contrast to the version of the meta-model I, this time the concrete event queue is contained in the model and therefore the statechart is evaluated in a more detailed context. The rule extracts the first element *e* of the event queue *events* and returns it. Note that the *selectEvent* rule of the meta-model II is indeed a refinement of that one of the meta-model I, because for any event *e* stored as first element in the event queue *events* will hold $e \in acceptable_events$. This refinement only holds w.r.t. safety properties, because the abstraction does not include the question whether an event occurs at all.

The rather abstract view of each component provided by the meta-model I occurs during the system design, when the internal component behavior is designed. For the overall architecture and coordination between shuttles and the rest of the system at least the possible event processing has to be added. While in the first case the supported abstraction levels permit to verify each object independently as an autonomous process reacting to the stimuli of its environment, the more detailed view requires that the process for a reasonable subset of all objects are embedded into a network of interconnected, communicating processes. Therefore, the processes will be mapped to a set of distributed ASM agents which are interconnected (see Agent parameter *a* of *selectEvent* in Figure 6 right-hand-side). At the more detailed levels the communication is therefore refined by replacing the non-deterministic occurrences of event by explicit managed event queues. In the end, we refine the base ASM meta-model by taking into account all implementation details, such as objects and their hierarchy, action execution semantics, guard evaluation against object attributes and their values. All these features extend our basic ASM, i.e. its data structure by defining new ASM domains and functions and its operation set in terms of new ASM rule definitions.

Other ASM meta-models may take other details into account that cover the dynamic semantics of statecharts and the non active local OO data structures and would therefore allow verifying properties which depend on the particular objects and their attribute values. Also, the communication can be refined by additional ASM agents that cover the more sophisticated behavior of communication channels. Note, we will adapt parts of the already existing ASM semantics for SDL [6]. The resulting lattice of different abstraction levels w.r.t. refinement permits to verify specific properties at an appropriate abstraction level. These different views also

reflect the transition from an architectural design (coarse-grain design) to a fine-grain-design and the details of an implementation.

4. VALIDATION

The design phase is characterized as the alternation of the construction and evaluation activities. To allow the designer to evaluate his or her system model in practice, besides verification validation in form of simulation and testing is common. To also support the evaluation of abstract and incomplete designs, we can use the same lattice of abstractions built by our ASM meta-models. Simulation of our ASM generated models is possible, because we use the AsmL [9] specification language, which supports the transformation of AsmL specifications into C++ code. Therefore, the executable ASM models permit the designer to validate his or her intermediate designs using the same abstraction levels presented for the verification.

When the final system should be simulated including all implementation details, it is however not useful to use a full ASM formalization with all its inherent overhead. Instead, the execution and debugging of the generated code is more appropriate (cf. [10]). Such an approach closes the gap between the simulation system (interpretation) and the software running on the real system following the "test what you run, run what you test" philosophy. Our attempt is to use the same code both, for the simulation as well as for the running system. So, the generated code has to be free from any kind of debug information and could be optimised for *special* issues, i.e. speed or space optimisations. To observe the running system, we developed a graphical debugging tool called 'Dobs' (Dynamic Object Browsing System). Dobs is able to display the internal object structure of a running Java virtual machine by various graphical representations [11].

5. RELATED WORK

The verification of software by means of model-checking is usually only feasible, when instead of the full system model of the software system an abstraction with reasonable state space size is considered. This process usually happens in an additional abstraction step that precedes the application of model-checking tools (cf. [12]). Therefore, experimenting with multiple abstraction levels to identify a one, which provides the required feasible model containing the relevant properties, is rather cumbersome without tool support.

Several projects exist, that deal with SDL or UML based design and verification by means of model-checking (e.g., [13][14][15][16][17]). For

example, an approach dealing with model-checking SDL specifications can be found in [13]. It also transforms the SDL code into an intermediate representation which can be further verified using a model-checker. However, these approaches, in contrast to the presented work, support only a single abstraction level.

Concerning simulation as the validation method, in contrast to other tools and simulation environments, e.g. STATEMATE [18], PROGRES [19], which simulate the specified model like an interpreter, our approach is to generate source code out of the specification and observe the running system.

In commercial sphere, there are already some tools available that allow the user to check his or her UML or SDL specification on a very high abstraction level. The case tool Telelogic TAU [20] is an example for the practical use of different abstraction levels for the validation of SDL specifications. On the highest abstraction level for example, only atomic transitions between the states of a process diagram are considered. The validation can then be refined by considering more details, like intermediate states or signals that take an undefined amount of time for their transmission. However, the focus of TAU lies on the verification of SDL specifications with limited possibilities concerning complex object structures and their modification.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented an approach, of how to construct and evaluate distributed and embedded systems. The construction of the system is realized by using a well defined design language, which employs a subset of UML and SDL. The presented support for multiple levels of abstractions by means of different ASM meta-models enables the validation and verification during the ongoing design. Therefore, it permits analysis of even incomplete models of complex distributed embedded system software and thus helps to cope with the state-space explosion problem. The presented approach is further embedded into the overall design methodology developed within the ISILEIT project for distributed production control systems.

REFERENCES

[1] Integrative Specification of Distributed Control Systems for the Flexible Automated Manufacturing (ISILEIT), German Research Foundation (DFG) program "integrative specification of engineering applications".: http://www.upb.de/cs/isileit/

[2] Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, Reading, Massachusetts, 1999.

[3] ITU-T Recommendation Z.100, Specification and Description Language (SDL), International Telecommunication Union (ITU), Geneva, 1994 + Addendum 1996.

[4] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In Proc. of the 22th Int. Conf. on Software Engineering (ICSE), Limerick, Irland. ACM Press, 2000.

[5] Y. Gurevich: Evolving Algebras 1993: Lipari Guide; E. Börger (Eds.): Specification and Validation Methods; Oxford University Press, 1995.

[6] R. Eschbach, U. Glässer, R. Gotzhein, M. von Löwis and A. Prinz: Formal Definition of SDL-2000 - Compiling and Running SDL Specifications as ASM Models. Journal of Universal Computer Science (J.UCS), October 2001.

[7] G. del Castillo and K. Winter: Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, Proc. on 6th Int. Conf. TACAS 2000, volume 1785 of LNCS, pages 331-346, 2000.

[8] H. Giese, M. Kardos and U. Nickel: Integrating Verification in a Design Process for Distributed Production Control Systems. Second International Workshop on Integration of Specification Techniques for Applications in Engineering (INT 2002). Grenoble, France, April 2002. (to appear)

[9] http://www.research.microsoft.com/fse/AsmL/default.html

[10] James D. Arthur, Markus K. G"oner, Kelly I. Hayhurst, and C. Michael Holloway. Evaluating the Effectiveness of Independent Verification and Validation. IEEE Computer, 32(10):79-83, October 1999.

[11] U. Nickel and J. Niere, 'Modelling and Simulation of a Material Flow System', in Proc. of Workshop 'Modellierung' (Mod), Bad Lippspringe, Germany, Gesellschaft für Informatik, 2001.

[12] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model Checking Large Software Specifications. IEEE Transactions on Software Engineering, 24(7):498-520, 1998.

[13] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, J. Sifakis. IF: An Intermediate Representation for SDL and its Applications. Proceedings of SDL-FORUM 1999 (Montreal, Canada) June 1999..

[14] Gihwon Kwon. Rewrite Rules and Operational Semantics for Model Checking UML Statecharts. In Andy Evans, Stuart Kent, and Bran Selic, editors, Proceedings of the third International Conference on the Unified Modeling Language (UML 2000), York, UK, volume 1939, page 528ff. Springer Verlag, October 2000.

[15] Johan Lilius and Iván Porres Paltor. vUML: a Tool for Verifying UML Models. In Proceedings of the 14th IEEE International Conference on Automated Software Engineering, Cocoa Beach, Florida, USA, 1999.

[16] Prasanta Bose. Automated Translation of UML Models of Architectures for Verification and Simulation Using SPIN. In Proceedings of the 14th IEEE International Conference on Automated Software Engineering, Cocoa Beach, Florida, USA, 1999.

[17] http://www-omega.imag.fr/.

[18] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Tauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems, IEEE Trans. Soft. Eng., 16, 403-414, 1990, Paderborn, 1999.

[19] A. Schürr, A. J. Winter, A. Zündorf. Graph grammar engineering with PROGRES. In W. Schäfer, Editor, Software Engineering - ESEC '95, LNCS 989, Springer Verlag, 1995.

[20] Telelogic Tau SDL Suite: http://www.telelogic.com