Bridging the Gap: Discrete-Event Systems for Software Engineering (Short Position Paper)

Juergen Dingel
School of Computing
Queen's University
Kingston, ON
dingel@cs.queensu.ca

Karen Rudie
Dept. of Elec. & Comp. Eng.
Queen's University
Kingston, ON
karen.rudie@queensu.ca

Chris Dragert
School of Computer Science
McGill University
Montreal, QC
chris.dragert@mail.mcgill.ca

ABSTRACT

Discrete-Event System Theory (DES) allows the automatic control of a system with respect to a specification describing desirable sequences of events. It offers a large body of work with strong theoretical results and tool support. In this paper, we advocate the application of DES to software engineering problems. We summarize preliminary results and provide a list of directions for future research.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming; D.2.2 [**Design Tools and Techniques**]: General; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Specification Techniques

General Terms

Design, Reliability, Verification

Keywords

Discrete-Event Systems, Model-Driven Development, Automatic Code Generation, Concurrency, Control Theory

1. INTRODUCTION

In [15], Henzinger and Sifakis lament the "wall between computer science and electrical engineering" which has held the "potential of embedded systems at bay". They point out that hardware system designers and software system designers typically use different kinds of models. In this paper, we suggest first steps towards realizing Henzinger and Sifakis' vision of "a new scientific foundation [...] which will ensure a systematic and even-handed integration" of computer science (CS) and electrical engineering (EE). However, we draw attention to a field of EE in which the models used are already very similar to those used in CS: In Discrete-Event System Theory (DES) systems are characterized by state changes which are caused by the occurrence of discrete events rather than the passage of time. In standard DES [28], finite state machines (FSMs) are used to specify the system G and the permissible event sequences E; then, a supervisor S is automatically generated such that the closed loop system S/G only exhibits behaviours inside E.

The main contribution of this paper is the observation that many problems in software engineering (SE) can be reduced to observing events of a system and restricting its behaviour to specific sequences. We provide preliminary evidence that DES offers solutions to many SE problems with impressive theoretical foundations and guarantees.

As a motivating scenario, consider the classical problem of synchronizing a collection of parallel threads such that some shared resource R is accessed in a mutually exclusive and deadlock-free fashion. After identifying the *relevant events* in each thread T_i, that is, the actions representing the start and the finish of an access to R, the control flow graph of T_i can be used to construct an FSM F_i describing the set of event sequences possible for T_i. The composition of all F_i will form G. The specification E is given by an FSM that accepts a sequence of start and finish events iff they satisfy mutual exclusion. Standard DES can be used to generate a supervisor S such that the composition of G and S is deadlock-free and exhibits only behaviours satisfying E; moreover, S is guaranteed to be minimally restrictive, i.e., S will prohibit *only those* behaviours of G which will result in a deadlock or violation of mutual exclusion. The problem of controlling concurrent programs is well-researched. However, the use of DES is still attractive because a provably correct solution is automatically generated, and, most importantly, that solution is guaranteed to not impose any unnecessary restrictions.

Our recent work and that of a few other researchers suggests that DES may offer rigorous solutions to many SE problems. Moreover, there appear to be interesting similarities between DES and model checking and some existing synthesis approaches.

We will first give a very brief overview of DES. Next, we will review preliminary results and then list potential additional areas of application and research.

2. WHAT IS DES?

Standard DES originated with Ramadge and Wonham in the 1980s [28]. FSMs are used to model the *system*, also called the *plant* G. Transitions in G are events, and the generated language is referred to as the *behaviour* of G. An FSM is called *nonblocking* if all states are reachable and a final state can always be reached. Under the assumption that the system is not considered deadlocked in a final state, nonblocking implies deadlock-freedom and livelock-freedom. Each event is either *controllable*, meaning it can be disabled, or *uncontrollable*, meaning it cannot or should not be prevented from occurring. For example, "send message" is typically a controllable event, while "message transmit fail" is an uncontrollable event. A specification, given as an FSM, provides the desired behaviour of the plant, and is called the *legal language* E. A plant G is called *controllable* with respect to a specification E if, for any string s from the prefix closure of E, there are no uncontrollable events e that could be generated by G at the state reached by s such that se would not be in the prefix closure of E. In other words, if something cannot be prevented, it must be legal.

In *supervisory control*, a supervisor S (also an FSM) is introduced to control the plant by enabling and disabling events based upon the events that occur in the plant. Figure 1 shows the basic relation between supervisor and plant.

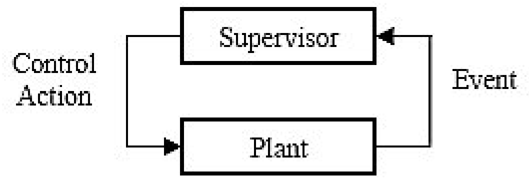

Figure 1: Supervisor-Plant Relation

Given a plant G and a legal language E, standard DES *supervisor synthesis* automatically generates a supervisor S such that the *closed-loop system*, S/G, given by the synchronous product of S and G, is guaranteed to be nonblocking and recognizes the largest controllable subset of the legal language.

The supervisor S guarantees not only deadlock-freedom and adherence to the specification E, but also minimal restrictiveness[1]. These strong theoretical guarantees set DES apart from most alternative approaches to related problems (e.g., generation of concurrency controllers [8], component adaptors [38], or resource managers [7]). DES supervisor synthesis is computable in polynomial time with respect to the product of the state sizes of G and E. However, just like model checking, DES supervisor synthesis suffers from the *state-space explosion* problem: The size of the plant may be exponential in the number of processes.

Many extensions of standard DES exist. For instance, *modular supervisory control* [42], allows for multiple specifications and multiple supervisors to act independently on a single plant. However, for the composite system to be nonblocking, the constituent modules must satisfy additional criteria. In *partial observation* [24], supervisors are no longer assumed to be able to observe *all* events that the plant generates but instead must base their control decisions on only partial information. In *hierarchical DES* [44] information is aggregated so that only crucial information is seen by a higher-level coordinator whose decisions then impact a lower-level controller. In *decentralized DES* [6, 30], multiple supervisors, each of which only has partial observations and partial control, perform separate actions so that together some overall goal is enforced. Standard DES has also been extended to include *real-time constraints* [3], *probabilities* [37] and *dynamic* systems, whereby the plant may vary over time [13]. For a more detailed description of DES the reader is referred to [4].

Most industrial applications of DES appear to be in the context of manufacturing systems [2, 43, 23] but other applications include communication systems [29] and failure diagnosis [32]. Many DES tools exist (e.g., [35, 19, 9, 34]).

[1]Note that the language recognized by S/G will be empty if a deadlock or specification violation is unavoidable; in that case, the supervisor will prevent G from executing at all.

3. PRELIMINARY RESEARCH RESULTS

To the best of our knowledge, so far DES has been applied to only five domains related to software: analysis of database transactions, protocol verification, feature interaction, execution of workflows and concurrent software.

DES for analysis of database transactions.

In [22], standard DES is used for comparing the performance of different database transaction execution protocols.

DES for protocol verification.

In [29], decentralized DES control is used to model agents in a telecommunication network. Results from [30] are used to detect errors in a protocol at the data link layer of a communication network. We believe that comparable techniques could be used to do verification at other levels of a distributed system.

DES and feature interaction.

Modular supervisory control and partial observation are brought to bear on the problem of feature interaction in telecommunication services [36]. Services and features for multiple subscribers in a telephone network are modelled as FSMs. Negative interactions between features can be captured by the kinds of conflicts automatically detected among modular DES supervisors. Consequently, the work on conflict resolution in DES seems relevant here [41].

DES for execution of IT workflows.

The use of DES for the safe execution of IT automation workflows is described in [40]. The generated supervisor ensures that the execution of a, for instance, BPEL workflow does not deadlock or visit any "forbidden" states specified by the user.

DES for concurrent programs.

Two research teams have independently applied DES for the generation of controllers for concurrent software [39, 11]. The motivating scenario is very similar to the one sketched in the introduction, that is, the supervisor controls the execution of a collection of concurrently executing threads. In [39], the system model G is obtained by translating the control flow graph (CFG) of each thread into a Petri net and composing the results. A technique called *supervisory control based on place invariants* is used to construct a supervisor that ensures the deadlock-free execution of the threads. Validation is carried out by executing randomly generated concurrent programs. Our own work [10, 11] targets the same scenario and also leverages the CFGs for the construction of G. However, standard DES is used. Moreover, the user can supply a specification E (in the form of an FSM) that every sequence of events exhibited by the concurrent program must satisfy. The concurrent program is assumed to be marked up manually with the events mentioned in the specification. Binary semaphores are inserted just before every controllable event and used by the supervisor to enable and disable events. The Java PathFinder model checker is used to validate that the system augmented by a Java implementation of the supervisor behaves as desired. An overview the process presented in [10, 11] is given in Figure 2.

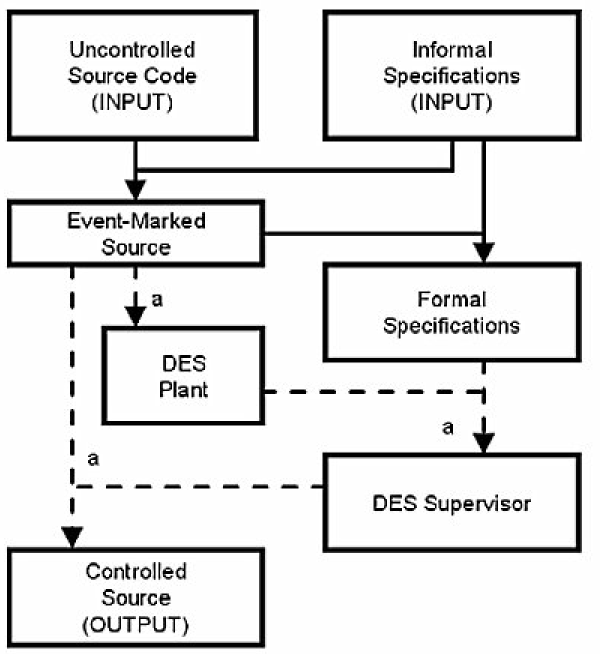
Uncontrolled Source Code (INPUT)
Informal Specifications (INPUT)
Event-Marked Source
a
Formal Specifications
DES Plant
a
a
DES Supervisor
Controlled Source (OUTPUT)

Figure 2: Process to create and inject concurrency controls into source code (automated steps are marked with 'a')

As future work, we are investigating the use of μ-calculus-based DES [45] to allow for the specification of liveness properties. Moreover, neither approach supports dynamic thread creation; results from dynamic DES [13] may be helpful here.

4. ADDITIONAL AREAS OF APPLICATION AND RESEARCH

DES provides a mechanism to enforce event orderings. Consequently, at least the following potential areas of application and research suggest themselves.

Component based software engineering.

The goal behind attaching behavioural specifications to interfaces is to describe permissible uses of the interface to facilitate component composition and enable modular reasoning. Examples for approaches supporting behavioural interface constraints can be found on the programming languages level (e.g., Java Modeling Language (JML) [5], Eiffel [25], or session types [16, 17]), on the middleware level (e.g., Corba IDLs), on the software architecture level (e.g., behaviour protocols [27]), on the modeling level (e.g., UML's Protocol State Machines [26]), and even on the operating systems level [18, 33]. Interestingly, many of the specification formalisms used are similar to FSMs and describe permissible sequences of events (i.e., interface accesses).

Work leveraging these kinds of interface specifications includes Microsoft's SDV [1], and work on component adaptation and assembly [38]. In [38], a coordinator is generated that guarantees the deadlock-free and specification-violation-free execution of a collection of components which are assumed to adhere to a particular architectural style and to have syntactically matching interfaces. The approach does not use DES, but some noteworthy similarities exist. Models and specifications are given as FSMs (finite labelled transition systems). Moreover, while the details still remain to be checked, the coordinator construction appears similar to the supervisor synthesis in DES, using, for instance, a notion of "last chance states", that is, states which present the last opportunity to prevent a deadlock.

We believe that an application of DES to the problem of checking and enforcing behavioural interface specifications is very promising. Relevant extensions of standard DES include hierarchical, decentralized and dnynamic DES.

Run-time monitoring, steering and adaptive computing.

The work on run-time monitoring and steering (e.g., [14, 21, 20]) anticipates current work on adaptive and self-managing systems. In standard DES, the system model is fully known and the synthesis algorithm uses it to construct a supervisor that is prepared for all possible system behaviours. Software monitors on the other hand typically do not have full knowledge of future system behaviours. *Supervisory control with limited lookahead* [31, 13] deals with this situation by allowing the supervisor to be constructed dynamically ("on-the-fly") based on limited information about the future behaviour of the system. We suggest investigating the applicability of this DES extension to adaptive and self-managing systems. Interestingly, the application of steering to DES has already been explored [12].

DES and other analysis and synthesis techniques.

The theoretical and practical similarities and differences between the various forms of model checking and supervisor synthesis are worth investigating. Techniques from μ-calculus model checking are used in [45] to develop a version of DES that supports μ-calculus specifications. However, we are not aware of any work attempting to leverage DES for model checking. An intriguing vision is the use of supervisor synthesis to enable model checkers to not only to find bugs but also corresponding bug fixes.

Other approaches to supervisor or controller synthesis exist. For instance, the work in [38] has already been mentioned. Moreover, game theory has been used to generate deadlock-avoiding resource managers [7]. Comparing these approaches with DES may be very fruitful.

5. CONCLUSION

In this paper, we argue that researchers in DES and SE should become much more aware of each other's problems and techniques, because this would not only benefit both fields but also help bridge the gap between EE and CS. Apart from the more concrete research problems outlined above, more general, overarching research challenges include: Which parts of DES are most suitable for which kinds of SE problems? Given that DES also is prone to the state explosion problem, which optimization and implementation techniques ensure sufficient scalability? How useful are techniques from, e.g., hierarchical and decentralized DES or optimistic, non-blocking concurrency to this end? How can the application of DES to a specific problem be simplified via automatic techniques that, e.g., extract models from code or prepare code for supervisory control through instrumentation (as in, e.g., [21])?

6. REFERENCES

[1] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, Portland, OR, January 2002.

[2] B. A. Brandin. The real-time supervisory control of an experimental manufacturing cell. *IEEE Transactions on Robotics and Automation.*, 12:1–14, 1996.

[3] B. A. Brandin and W. M. Wonham. The supervisory control of timed DES. *IEEE Transactions on Automatic Control*, 39(2):329–342, 1994.

[4] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems, 2nd edition.* Springer, 2008.

[5] P. Chalin, J. Kiniry, G. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. In *Formal Methods for Components and Objects (FMCO'05)*, LNCS 4111, pages 342–363, 2006.

[6] R. Cieslak, C. Desclaux, A. S. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, 1988.

[7] L. de Alfaro, M. Faella, R. Majumdar, and V. Raman. Code aware resource management. In *5th International Conference on Embedded Software (EMSOFT'05)*, pages 191–202, 2005.

[8] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *24th International Conference on Software Engineering (ICSE'02)*, pages 442–452, 2002.

[9] DESUMA. Discrete Event Systems Group, University of Michigan, `www.eecs.umich.edu/umdes/toolboxes.html`, Dec. 2008.

[10] C. Dragert. Generation of concurrency controls using discrete-event systems. Master's thesis, School of Computing, Queen's Univ., Sept. 2008.

[11] C. Dragert, J. Dingel, and K. Rudie. Generation of concurrency control code using discrete-event systems theory. In *16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'08)*, Nov. 2008.

[12] A. Easwaran, S. Kannan, and O. Sokolsky. Steering of discrete event systems: Control theory approach. In *ENTCS*, volume 144, pages 21–39, 2005.

[13] L. Grigorov and K. Rudie. Near-optimal online control of dynamic discrete-event systems. *Discrete Event Dynamic Systems*, 16(4):419–449, 2006.

[14] W. Gu, G. Eisenhauer, and K. Schwan. Falcon: On-line monitoring and steering of parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, Aug. 1998.

[15] T. Henzinger and J. Sifakis. The discipline of embedded systems design. *IEEE Computer*, 40(10):32–40, 2007.

[16] K. Honda. Types for dyadic interaction. In *International Conference on Concurrency Theory (Concur'93)*, pages 509–523, 1993.

[17] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming (ESOP'98)*, LNCS 1381, pages 122–138, 1998.

[18] G. Hunt and J. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[19] IDES: The integrated discrete-event systems tool. Discrete-Event Control Systems Lab, Queen's Univ., `qshare.queensu.ca/Users01/rudie/www/software.html`, Dec. 2008.

[20] Y. Jia and J. M. Atlee. Run-time management of feature interactions. In *ICSE Workshop on Component-based Software Engineering.*, May 2003.

[21] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, checking, and steering of real-time systems. In *2nd International Workshop on Run-time Verification (RV'02)*, July 2002.

[22] S. Lafortune. Modeling and analysis of transaction execution in database systems. *IEEE Transactions on Automatic Control*, 33(5), 1988.

[23] R. J. Leduc, M. Lawford, and P. Dai. Hierarchical interface-based supervisory control of flexible manufacturing system. *IEEE Transactions Control Systems Technology*, 14(4):654–668, 2006.

[24] F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Science: An International Journal*, 44(3):173–198, 1988.

[25] B. Meyer. *Object-Oriented Software Construction (2nd Edition).* Prentice Hall, 1997.

[26] Object Management Group. *UML Specification, Version 2.1.2 (formal/07-11-01)*, 2007.

[27] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[28] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[29] K. Rudie and W. M. Wonham. Protocol verification using discrete-event systems. In *31st IEEE Conference on Decision and Control*, pages 3770–3777, 1992.

[30] K. Rudie and W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.

[31] S.-L.Chung, S. Lafortune, and F. Lin. Limited lookahead in supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, 37(12):1921–1935, 1992.

[32] M. Sampath, R. Sengupta, S. Lafortune, K.Sinnamohideen, and D. Teneketzis. Diagnosibility of discrete event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, 1995.

[33] Z. Stengel and T. Bultan. Analyzing singularity channel contracts. Accepted for publication in the 2009 International Symposium on Software Testing and Analysis (ISSTA 2009).

[34] Supremica. Department of Signals and Systems, Chalmers University of Technology, `http://www.supremica.org/`, Dec. 2008.

[35] TCT. Systems and Control Group, Deptartment of Electrical and Computer Engineering, University of Toronto, `www.control.toronto.edu/DES`, Mar. 2008.

[36] J. G. Thistle, R. P. Malhamé, and H.-H. Hoang. Feature interaction modelling,detection and resolution: A supervisory control approach. In *Feature Interactions in Telecommunications and Distributed Systems IV*, pages 93–107, 1997.

[37] D. Thorsley and D. Teneketzis. Diagnosability of stochastic discrete-event systems. *IEEE Transactions on Automatic Control*, 50(4):476–492, 2005.

[38] M. Tivoli and P. Inverardi. Failure-free coordinator synthesis for component-based architectures. *Science of Computer Programming*, 71(3):181–212, May 2008.

[39] Y. Wang, T. Kelly, M. Kudlur, S. Mahlke, and S. Lafortune. The application of supervisory control to

deadlock avoidance in concurrent software. In *9th International Workshop on Discrete Event Systems (WODES'08)*, May 2008.

[40] Y. Wang, T. Kelly, and S. Lafortune. Discrete control for for safe execution of it automation workflows. In *European Conference on Computer Systems (EuroSys'07)*, 2007.

[41] K. C. Wong, J. G. Thistle, R. P. Malhamé, and H.-H. Hoang. Supervisory control of distributed systems: Conflict resolution. *Discrete Event Dynamic Systems*, 10(1-2), Jan. 2000.

[42] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals, and Systems*, 1:13–30, 1988.

[43] A. Yalcin and T. O. Boucher. Deadlock avoidance in flexible manufacturing systems using finite automata. *IEEE Transactions on Robotics and Automation*, 16(4):424–429, August 2000.

[44] H. Zhong and W. M. Wonham. On the consistency of hierarchical supervision in discrete-event systems. *IEEE Transactions on Automatic Control*, 35(10):1125–1134, 1990.

[45] R. Ziller and K. Schneider. Combining supervisor synthesis and model checking. *Transactions on Embedded Computing Systems*, 4(2):221–362, May 2005.