Proc. Estonian Acad. Sci. Phys. Math., 1998, **47**, 3, 186–197
https://doi.org/10.3176/phys.math.1998.3.04

MSCPDL – A LANGUAGE FOR BEHAVIOURAL ABSTRACTION

Mikkel CHRISTIANSEN

Department of Computer Science, Aalborg University, Fredrik Bajers Vej 7 E1, DK-9220 Aalborg Øst, Denmark; e-mail: mixxel@cs.auc.dk

Received 19 January 1998

Abstract. Distributed systems consist of a set of processes. These processes cooperate for a common goal through the use of a communication medium. Gaining an understanding of how a distributed system works is often difficult and complex. This complexity is introduced by the communication in the system because it is necessary to understand both the single processes and how it interacts with the rest of the system.

A method for presenting the complexity of a distributed system is the use of message sequence charts (MSC). A MSC gives a graphical presentation of a single execution of a distributed system. Even with this restriction, MSCs easily grow large and incomprehensible. This motivates the need for tools that allow developers to analyze MSCs.

This paper presents two main results from a project dedicated to building a tool for analyzing MSCs. The aim of the tool was to reduce the complexity of extracting information from a MSC. The first main result is a formal description of MSCs. The second result is a language for describing patterns in a MSC. The tool has been realized in a prototype implementation.

Key words: message sequence charts, debugging, distributed systems, languages.

1. INTRODUCTION

Understanding the details of how a distributed system works is often difficult. In general, this difficulty is caused by the event-based architecture of distributed systems. A distributed system consists of several communicating processes. Each process consists of a sequence of events. An event of a process may initially be triggered by an external input. This event may trigger a communication event, which again triggers an event in another process and so forth. As a consequence, the details of how a distributed system works are based on complex communication patterns between the processes of the system.

A traditional technique for displaying the internal details of how a distributed system works is the use of message sequence charts (MSCs). A MSC gives a graphical representation of a single execution for a distributed system. A number of tools related to the development of distributed systems use MSCs. Examples are XSPIN [1] and Objectime [2]. Furthermore, it should be mentioned that MSCs are subject to standardization by CCITT [3].

A problem with MSCs is that they easily grow large and incomprehensible. This problem is caused by the fact that a distributed system can consist of numerous processes each with complex communication patterns.

In this paper, a tool for analyzing MSCs is presented. The purpose of the tool is to support developers when analyzing large and complex MSCs. The tool allows one to use a language for specifying queries. The queries are then executed on a MSC for retrieving the specified information. The query language is called MSC pattern description language (MSCPDL) and the tool is called TRACEINVADER.

The paper is structured as follows. First, a number of general terms are defined. On these terms, the aim of the tool is refined. Then a formal definition of MSCs is given. This is followed by an informal description of the query language. This description is closely related to the definition of MSCs. Having presented the query language and MSCs, a few examples of use are given. The final part contains a conclusion and a discussion of possible future work.

2. DEFINITIONS

A *distributed system* consists of a set of communicating processes. These processes are distributed over a set of processors which are connected through some communication media.

The *behaviour* of any process in a distributed system is given by the events occurring in the process itself [4]. An event is an atomic action, i.e., an action that is independent of any event of any other process. Examples of events could be an assignment operation, a communication operation, or a calculation.

A *causal relationship* between two events of a distributed system is an indication that one event may have caused or influenced the other event. The *behaviour* of a distributed system is given by the events occurring in each process of the system and the causal relations between these events. Therefore the causal relations are valuable when analyzing the workings of a distributed system.

As mentioned in the introduction, a *MSC* is a technique for displaying single execution of a distributed system. This means that different executions of a system may result in different MSCs. The difference between executions can depend both on input to the system and on whether non-determinacy is possible in the specific implementation. Figure 1a is an example MSC, where the system consists of four processes. A node of a MSC represents an event and a directed arrow represents a causal relation. Each process is represented as a vertical line of consecutive causalities and events. The line grows downward as the time increases. The directed

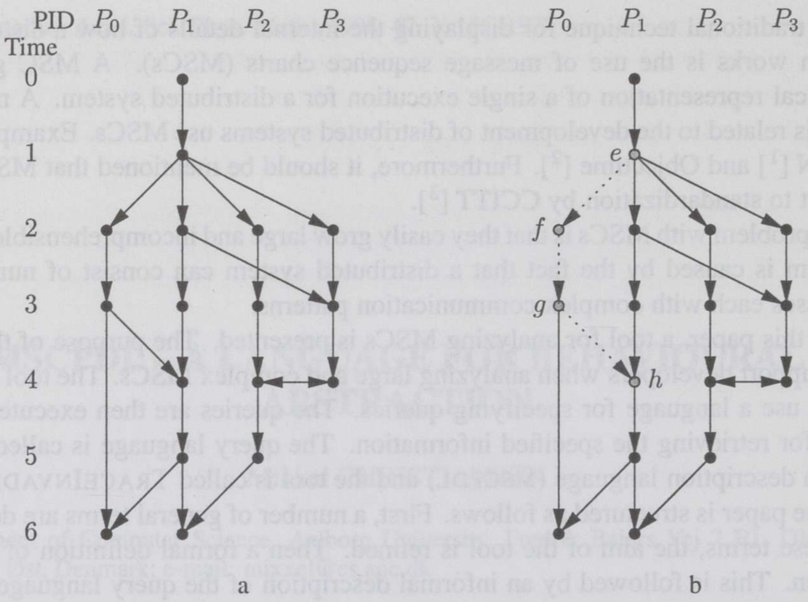

Fig. 1. A small MSC (a), a MSC where a client-server communication has been marked with dotted arrows (b).

arrows that interconnect processes represent asynchronous communication and the horizontal bidirectional arrows represent synchronizations. In Section 3, a formal definition of MSCs is given.

Patterns of behaviour describe a subset of a MSC. An example is the pattern describing an asynchronous communication. Then, each asynchronous communication in a MSC can be instantiated as an instance of the pattern. Another example is a pattern that describes a client-server communication. Here, a client requests a service from a server and the server responds with the result of the service. Figure 1b is a MSC where an instance of the client-server pattern has been marked with dotted arrows.

Based on these definitions, the purpose of the tool is refined. The objective of the tool is to reduce the complexity of extracting information from MSCs by the use of a simple language, called MSCPDL. The language is designed for describing the patterns of behaviour. Instances of the described patterns are found through an efficient search of a MSC.

3. MESSAGE SEQUENCE CHARTS

A MSC is a representation of a single execution for a distributed system. In this section, a definition of MSCs is given along with a number of operations used for comparing the nodes in a MSC. Furthermore, a technique for optimizing the evaluation of MSC operations is introduced.

The definition of MSCs is based on the observation that a MSC is graphs. Therefore the definition is given by the first defining of the terms node and edge. But first, an attribute environment is defined. This is a function for supporting storage and retrieval of information attached to a specific node:

Definition 3.1. *[Attribute environment] An attribute is a member of* $\mathbf{Name}$*, which is the set of strings consisting of alphanumeric letters. An attribute environment is a function in the domain* $\mathbf{Env_a} = \mathbf{Name} \hookrightarrow \mathbb{Z}$ *mapping from an attribute name to a value.*

A relevant use of the attribute environment is to store the line number of an event in the *lineno* attribute. This would allow one to relate the event of a node to a specific expression in the source code for an application.

Definition 3.2. *[Node] A node is identified by a unique number and has an attribute environment for storing attribute values. It is a 2-tuple* (i, env_a)*, where* $i \in \mathbb{Z}$ *and* $env_a \in \mathbf{Env_a}$*. The set of nodes is denoted by* $\mathbf{Node}$*.*

Each node of a graph has a unique identity i. This allows references to the specific nodes of the graph by using dot notation ($e.i$). A node of a graph resembles an event for a process.

Definition 3.3. *[Edge] An edge is a 3-tuple* $(src, dst, type)$*.* $\{src, dst\} \in \mathbf{Node}$ *and* $type \in \{0, 1\}$*. The type indicates whether this edge is directed or bidirectional. If* $type$ *is* 0*, then the edge is bidirectional, and if the* $type$ *is* 1*, then the edge is directed. The set of edges is denoted by* $\mathbf{Edge}$*.*

As with a node, dot notation is used for accessing the *type* of an edge. An edge can be both directed or bidirectional. If it is directed, then it represents a causality which can be both an internal causality of a process or an asynchronous communication between processes. If it is bidirectional, then it represents a synchronization between the two processes. Having defined nodes and edges, a full definition of a MSC can be given.

Definition 3.4. *[MSC] A MSC is a directed graph defined by the 2-tuple* (N, E)*, where* $N \subseteq \mathbf{Node}$ *and* $E \subseteq \mathbf{Edge}$*. The set of all MSCs is denoted by* G_c*.*

The nodes of a MSC are compared, using the boolean operator synchronization $(=)$*, causality* $(\rightsquigarrow)$*, concurrency* $(\|)$*, and minimal causality* $(\rightarrow)$*. Given two nodes* *a and b, the operators are defined as*

$$a = b \Leftrightarrow \sum_{e \in \mathbf{E}} e.type = 0$$

$$a \rightsquigarrow b \Leftrightarrow \sum_{e \in \mathbf{E}} e.type > 0$$

$$a \| b \Leftrightarrow \neg(a \rightsquigarrow b) \wedge \neg(a \rightsquigarrow b)$$

$$a \rightarrow b \Leftrightarrow \sum_{e \in \mathbf{E}} e.type = 1,$$

where $\{\mathbf{E} \mid$ *is the edges of the path between a and b*$\}$.

If graph g is built by inserting a directed edge for each minimal causality ($\rightarrow$) *from a MSC, then g will be an acyclic digraph.*

The final property on a MSC ensures that no message can go back in time. Furthermore, the causality operator ($\rightsquigarrow$) is transitive and not reflexive, meaning that the situation $a \rightsquigarrow b$ and $b \rightsquigarrow a$ cannot occur.

In terms of implementation, the operations $=$, $\rightsquigarrow$, and $\|$ are quite inefficient. The inefficiency is caused by the use of paths in the operations. Finding a path between two nodes in a MSC is the worst case $O(n)$, where n is the number of nodes of the MSC. By introducing vector time [5], these operations can be optimized significantly at the price of memory usage.

Vector time is a technique where each event in the MSC is stamped with a time vector. The value of the vector increases as events occur, which means that no vector is ever decremented. By comparing these vectors, using the two operators defined in [5], one can efficiently evaluate comparisons, using the operators $=$, $\rightsquigarrow$, and $\|$. Vector time is defined as follows:

Definition 3.5. *[Vector time] A time vector is a 2-tuple* $(i, t) \in \mathbf{VT} = \mathbb{Z} \times \mathbb{Z}^n$, *where t is a n-tuple indicating the knowledge of the n clocks in a system and i is the index of the local clock. The dimension of the time vector is n.*

Defined on a time vector are the boolean operators $=$, $\rightsquigarrow$, *and* $\|$. *Given two vector times* $v_1 = (i_1, (t_{1,1}, \ldots, t_{1,n}))$ *and* $v_2 = (i_2, (t_{2,1}, \ldots, t_{2,n}))$, *the operators are defined as:*

$$\begin{aligned} v_1 = v_2 &\Leftrightarrow t_{1,i} = t_{2,i} \text{ for } i = 1, \ldots, n \\ v_1 \rightsquigarrow v_2 &\Leftrightarrow t_{1,i_1} \le t_{2,i_1} \wedge t_{1,i_2} < t_{2,i_2} \\ v_1 \| v_2 &\Leftrightarrow \neg(v_1 \rightsquigarrow v_2) \wedge \neg(v_2 \rightsquigarrow v_1), \end{aligned}$$

where $=$ *checks for synchroneity,* $\rightsquigarrow$ *for causality, and* $\|$ *for concurrency.*

A description of the algorithm for constructing vector time on a MSC is given in [6].

4. MSCPDL LANGUAGE

In this section, an informal presentation of the language MSCPDL is given. First, the ideas and requirements of the language are stated. Next, the language is presented. The presentation is followed by two examples which illustrate the expressiveness of MSCPDL. A full operational semantics of the language is given in [6].

A definition of MSCs is given in the final section, where a simple language for extracting information from a MSC is described. The idea is to extract information from a MSC by first describing the patterns of behaviour and then searching for instances of these patterns.

Describing complex patterns of behaviour requires a mechanism for allowing abstraction. Otherwise, the work needed for describing a complex pattern will be unreasonable. Furthermore, the language should focus on describing the patterns of behaviour and not on how the search for patterns is done. With these two goals, inspired by Prolog [7] and general language principles, MSCPDL was designed.

4.1. Rules

Patterns are generally described through a number of rules. Each rule consists of a name, an interface, and a disjunction with one or more expressions. Each expression of a rule is a pattern description. This means that an instance of the rule is found if one of the expressions is matched. A rule has the following form:

$$r(args) \Leftarrow E_1|E_2|\ldots|E_n;$$

This declaration binds the rule r with the set of formal arguments $args$ to the expressions E_1 to E_n. Each expression of the rule has a private scope.

Expressions are build using three different groups of expressions: simple expressions, combined expressions, and rule invocations. In the following, these groups of expressions will be discussed.

4.1.1. Simple expressions

The simple expressions of an expression mostly correspond the different boolean operators that were defined on a MSC 3.4, only the $\rightsquigarrow$ operator has been omitted.

Causality expression ($e \rightarrow f$): Models the existence of a causality in a MSC, between nodes e and f of the MSC.

Sync expression ($e \leftrightarrow f$): Models the existence of synchronous relationship between nodes e and f of the MSC.

Parallel expression ($e \| f$): Models the existence of parallel relationship between nodes e and f of the MSC.

In these expressions, e and f are variables. An instance of a pattern described by a single simple expression may include both edges and nodes, depending on which expression is used. An example is the pattern described by the expression $e \to f$. An instance of this pattern will both describe the two nodes e and f and how they are causally related. But an instance of the pattern $e \| f$ will only include the two endpoints, since there is no causal relation to include.

As explained in the definition of a node, each node of the MSC has a set of attributes. The language allows the utilization of attributes by allowing comparisons. Nodes of the MSC can be compared by comparing their attributes with the attributes of other nodes or constant values. Syntactically, dot notation is used for referring to a value of a specific attribute.

Comparison expression ($C_i \ \theta \ C_j$): Models the existence of a relation between attributes bound to events or a constant value.

C_i and C_j can be any attribute ($e.p_i$) and ($f.p_j$) or a constant n. θ can be any of $\{<, \leq, =, \neq, \geq, >\}$. An instance of a comparison is the one or two nodes that fulfill the comparison.

4.1.2. Combined expressions

Simple expressions are combined in conjunctive and disjunctive expressions. The order in which the simple expressions are placed in a conjunction or disjunction is unimportant for the semantics of the pattern description. When it comes to the evaluation of an expression, it is evaluated from left to right.

Disjunction expressions ($E_1 \vee E_2$): The disjunction of the two expressions E_1 and E_2.

Conjunction expressions ($E_1 \wedge E_2$): The conjunction of the two expressions E_1 and E_2.

The scope of a combined expression is the entire expression. This means that variables used in more than one expression of the conjunction or disjunction are shared. Shared variables provide a simple technique for binding the expressions of a combined expression together. Combined expressions are instantiated by instantiating each of the simple expressions of the combined expression.

4.1.3. Rule invocation expressions

The expressive power needed for expressing patterns of behaviour is fulfilled by allowing compositionally defined patterns of behaviour. Patterns can be described compositionally by supporting rule invocations. Rule invocations are treated equally to simple expression. This means that it is possible to describe recursively defined patterns of behaviour, such as the transitive closure between two nodes in the MSC.

Rule invocation expressions ($r(x_1, \ldots, x_n)$): The rule invocation of the rule r with the actual parameters $x_1, \ldots, x_n$.

Rules are invoked with static scopes. The following is an example of two rule definitions:

$$\begin{aligned} foo(e, f) &\Leftarrow e \rightarrow f. \\ bar(a, b) &\Leftarrow foo(a, b). \end{aligned}$$

In the example, the rules *foo* and *bar* are defined. The *bar* rule invokes the rule *foo*. When the rule *foo* is invoked, then a separate scope is created for the evaluation of the $e \rightarrow f$ expression. The *bar* rule will have access to the result through the actual parameters after the evaluation of $foo(a, b)$.

4.2. Queries

A query is the outermost expression of a pattern description. It is an expression that applies a combination of rules for giving a final pattern of behaviour. In terms of syntax, it is simply an expression terminated with a question mark.

An example is the query: $e \rightarrow f$? When this query is evaluated, all instances of the pattern $e \rightarrow f$ are found from a given MSC.

4.3. Syntax

Having given a description of the different parts of MSCPDL, the full syntax is given on BNF:

$$\begin{aligned} \langle Q \rangle &::= \langle D_r \rangle.\langle E \rangle? \\ \langle D_r \rangle &::= r(x_1, \dots, x_n) \Leftarrow \langle E \rangle.\langle D_r \rangle \mid \varepsilon \\ \langle E \rangle &::= e\,\theta_1\,f \mid \langle C_1 \rangle\,\theta_2\,\langle C_2 \rangle \mid r(x_1, \dots, x_n) \mid \langle E_1 \rangle \wedge \langle E_2 \rangle \mid \langle E_1 \rangle \vee \langle E_2 \rangle \\ \langle C \rangle &::= n \mid e.p, \end{aligned}$$

where $\theta_1 \in \{\rightarrow, \leftrightarrow, \|\}$, $\theta_2 \in \{<, \leq, =, \neq, \geq, >\}$, $n \in \mathbf{Num}$ and $\mathbf{Num}$ is the set of all integer numerals.

4.4. Evaluation

Queries are evaluated by searching a MSC for the instances of a specific pattern. A more precise description is as follows:

Definition 4.1. *[Evaluation] The evaluation function in* MSCPDL *is a function in the form:* $\mathbf{G_c} \hookrightarrow (\mathbf{Q} \hookrightarrow 2^{\mathbf{G_c}})$, *where* $\mathbf{Q}$ *is the set of all queries.*

This definition describes the evaluation function as a function that given a MSC and a pattern description, it returns a set of MSCs. This means that an instance of a pattern is a new MSC. Such an instance MSC consists of the nodes and edges described by the matched pattern. Additionally, a name space mapping is included. This name space maps variables of the query to nodes. Instantiating a pattern as a MSC allows one to continue with a search for patterns in the new MSC.

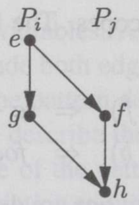

Fig. 2. The pattern of two successive communications.

4.5. Implementation

The described language and a search algorithm has been implemented in a prototype with a graphical userinterface. The prototype is called TRACEINVADER. In TRACEINVADER one can describe patterns of behaviour and search for instances of these in a MSC. Instances of queries are marked in the MSC, as shown in Fig. 1b. It should be mentioned that the prototype takes advantage of vector time as described in Definition 3.5.

5. EXAMPLES

In this section, two examples of pattern descriptions are given to illustrate the basic capabilities of MSCPDL.

5.1. Successive communications

The first example is a query that describes two successive communications from one process to any other process. Figure 2 is a MSC of the pattern described in the example.

Example 5.1.

$$Rules: \quad com(a,b) \Leftarrow a \rightarrow b \wedge b.pid \neq a.pid;$$

$$\begin{aligned} Query: \quad & com(e,f) \wedge e \rightarrow g \wedge e.pid = g.pid \wedge \\ & com(g,h) \wedge f.pid = h.pid?\ , \end{aligned}$$

where pid refers to the process identity attribute.

The example applies the rule mechanism for defining a general asynchronous communication. This results in the rule $com(a,b)$. Having defined an asynchronous communication, the query describes the two successive communications, using simple expressions and the $com(a,b)$ rule.

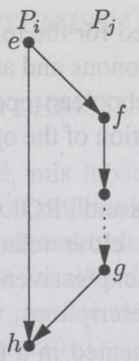

Fig. 3. The pattern of the client-server communication. The dotted arrow means zero or more edges and nodes.

5.2. Client-server communication

The next example uses recursion for describing a client-server communication. An instance of the pattern is shown in Fig. 3.

Example 5.2.

$$\begin{aligned}
Rules: \quad & com(a,b) \Leftarrow a \to b \wedge b.pid \neq a.pid;\\
& tc(a,b) \Leftarrow a \to b \wedge a.pid = b.pid \;\mid\\
& \qquad\qquad a \to c \wedge a.pid = c.pid \wedge c.type = \text{“}int\text{”} \wedge tc(c,b);\\
& clientserver(a,b,c,d) \Leftarrow com(a,b) \wedge tc(b,c) \wedge com(c,d) \wedge\\
& \qquad\qquad a \to d \wedge a.pid = d.pid;
\end{aligned}$$

$$Query: \quad clientserver(e,f,g,h)?,$$

where pid refers to the process identity attribute and the "type" refers the type of event. Events of the type "int" are events that do not involve any communication.

The $tc(a,b)$ rule uses recursion to find the transitive closure between two nodes. In this case, an *invariant* has been put on the elements in the closure. The invariant says that events of the closure should be of the type *internal* ($type = int$). The recursive part of the $tc(a,b)$ rule is evaluated as i PROLOG. First, the variables a and c are bound, then the rule is instantiated. The $clientserver(a,b,c,d)$ rule first describes a communication from one process to any other process. The $tc(a,b)$ rule is then used for describing any number of consecutive internal events in a server process.

6. CONCLUSIONS AND FUTURE WORK

The purpose of the tool described in this paper was to minimize the complexity of extracting information from a MSC by the use of a simple language.

A solid foundation was provided for the tool by a formal definition of a MSC. The definition supports both synchronous and asynchronous communication. In the definition of a MSC, a number of boolean operators were defined for comparing events of the MSC. Efficient evaluation of the operators was ensured by introducing a technique based on vector time.

Based on the definition of MSCs and PROLOG, a small but expressive language was designed. The language has close relation to MSCs, because it uses the operations defined on a MSC. The expressiveness of the language was ensured by allowing the rule-defined pattern descriptions, which can be invoked recursively.

The language has been implemented in a prototype tool. The tool provides a graphical userinterface by which it is possible to analyze a MSC by searching for a specific pattern of behaviour. The implementation takes advantage of vector time for ensuring an efficient search for patterns.

Optimization of the search for patterns described in MSCPDL is quite complicated because of the recursive nature of the patterns. A study of this issue is relevant because a developer needs freedom to describe a pattern, without worrying on how it is retrieved. Furthermore, it would be interesting to test the expressibility of MSCPDL in a larger context. In XSPIN [1], MSCs are generated as a result of simulating distributed systems. An idea is to analyze MSCs that are produced during these simulations.

ACKNOWLEDGEMENTS

My thanks are due to Jesper Langfeldt Hagen and Kristian Qvistgaard Skov.

REFERENCES

1. Holzmann, G. J. *Design and Validation of Computer Protocols*. Prentice Hall, N. J., 1991.
2. Selic, B., Gullekson, G., and Ward, P. T. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.
3. *CCITT. Recommendation z.120: Message sequence chart (MSC)*. Technical Report, CCITT, Geneva, 1992.
4. Bates, P. C. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, February 1995, **13**, 1, 1–31.
5. Fidge, C. Logical time in distributed computing systems. *Computer*, August 1991, **24**, 8, 29–33.
6. Christiansen, M., Hagen, J., and Qvistgaard Skov, K. *TraceInvader–A Tool for Debugging Distributed Applications*. Master's thesis. Department of Computer Science, Aalborg University, Aalborg Øst, Denmark, January 1997.
7. Sterling, L. and Shapiro, E. *The Art of Prolog*. MIT Press, Cambridge, USA, 1986.

MSCPDL – KÄITUMIST ÜLDISTAV KEEL

Mikkel CHRISTIANSEN

On defineeritud MSCPDL keel, mis lubab vähendada sõnumite järjestusdiagrammidest (MSC) informatsiooni ekstraktimise keerukust. MSCPDL võimaldab nn. käitumiskujunditena esitada abstraktselt MSC tippudevahelisi põhjuslikke seoseid. Keel on realiseeritud MSC analüüsikeskkonna prototüübina, mida saab kasutada MSCPDL-is kirjeldatud kujundite eksemplaride otsimiseks. Edasise töö eesmärk on selgitada analüsaatori praktilisi võimalusi võrguprotokollide analüüsil.