Proceedings of the 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION, Corfu, Greece, August 17-19, 2005 (pp541-546)

Relation between UML2 Activity Diagrams and CSP algebra

MSc. FRANTISEK SCUGLIK
Department of Information Systems
Brno University of Technology
Bozetechova 2, 612 36 Brno
CZECH REPUBLIC

Abstract: The computer systems grows from year to year and they impact our everyday live. Therefore, their failure is unacceptable. One of the techniques, how to increase reliability of a system, is the utilization of formal methods and verification. Unfortunately, most formal methods are mathematically based and system developers refuse to learn such new techniques. This paper aims to present a possibility how to automatically transform a UML activity diagram well known by most system developers into a CSP formal specification which can then be verified.

Key-Words: UML, CSP, Formal Specification, Verification, Automated translation

1 Introduction

In the past decades, the usage of computers and computer systems has grown incredibly. At present day, not only huge corporations use computer based systems but they impact every man's live. Almost each person in the world utilizes some type of computer based system. Moreover, with the successful expansion of internet the world has changed into one huge network where communications and businesses are taking place on-line continuously. Consequently to this expansion, it is no more possible for a single person to develop a complex computer based system. Instead, whole development groups counting up to hundreds people participate in the development process. Particular groups develop particular parts of the whole system and together they produce huge amount of code. Assuring that these parts will successfully work together is a difficult task. Besides, most of the processors in computer based systems are not single processor workstations but they are components of embedded systems and processor networks in complex control systems like car, train and airplane control systems. Such systems surround us every day and we often do not realize how much a failure can impact our lives. Therefore, these systems need to be reliable to guarantee their correct functionality.

System testing and verification can be applied in different stages of the software life cycle, but the earlier, the better. The earlier an error in the system is detected, the less damage it has done, and the cheaper is it to fix the error. Besides, the utilization of formal methods involves specification of systems, which are models of the real world. But real world applications are continuous therefore the model would be infinite. Because the verification process is limited in space, abstraction must be applied to the model. Therefore, the specification should include only that properties which must the system fit to be reliable. Construction of such specification represents a difficult topic. On the other hand, the more detailed specification of the critical parts the better. But the system specifications constructed by software engineers is often more detailed than the formal model. This inconsistency can lead to later errors. Therefore, there is a need for unified modeling methodology for both, the construction of formal model and the construction of understandable system specification model. Formal methods are most efficiently used within a project when properly used under such a methodology.

This contribution focuses on introducing formal methods into standardized software design methodology. More precisely, this paper describes the kernel of a developed tool for system designers. This tool connects the informal way of specifying systems using Unified Modeling Language (UML) and the formal algebra of Communicating Sequential Processes (CSP). Action diagrams from UML (precisely of second version of UML – UML2) are translated into formal model denoted in CSP. Such model can then be verified in common verification tools which are already developed. So, the formalism is hidden for the system engineer, and he does not need to learn new techniques but utilizes common UML.

The following section presents motivation for the development of this tool which translates UML Activity diagrams described in section 4 into CSP algebra introduced by Tony R. Hoare and denoted in

Proceedings of the 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION, Corfu, Greece, August 17-19, 2005 (pp541-546)

section 3. Section 5 invites relation between UML diagrams and CSP algebra.

2 Motivation

At present day, in real development processes it is still frequently that the formal specification and verification activities haves to compete with other development processes. To this situation contribute several factors:

- formal methods are relatively new and the developers do not have experience with use of it
- the development process inclines to be finished still quicker and it seems that formal methods require additional investments in time, money and human resources and the fact that formal methods can save time and money by finding errors is omitted
- the usage of mathematically based formal methods requires the user to educate himself in such technique instead of using a simply to use, user friendly interface which is still a rarity

When considering these claims, the main disadvantage of formal methods is the lack on a user friendly interface. Therefore, few goals for solving this situation were specified:

- do not develop new formalisms but utilize the already accessible ones
- do not press the user to learn new methods in software development cycle
- utilize common software development methodologies even for system formal specification
- hide the formalism in the background

3 Communicating Sequential Processes subset

Understanding, designing and building concurrent systems represent a major challenge for computer science. The involved complications differ from the sequential programming problems, therefore the concurrent systems requires systematic approaches.

Concurrent systems are all around us. They consist of independent, but communicating components. The familiar examples include:

- the network of bank cash machines
- the internet
- the telephone system
- the components of a PC

The algebra of CSP provides a possibility for concurrent systems to be modeled in more elementary and abstract way. It is supported by particular software tools which offer system analysis and verification.

CSP describes processes – objects which exist independent on each other, but may communicate. During their lifetime, processes can perform various actions or events. These events represent the visible part of modeled processes. For example, when describing a simple vending machine, two events may be interesting:

1. *coin* – represent insertion of a coin
2. *choc* – represents the appearance of a chocolate

The set of events used by the process to represent its behavior is called alphabet or interface. During the process activity the events in the interface may occur once, many times, or not at all. Which events should be included in the interface depends on aspects of process behavior which are interesting. For example, when specifying a lecture and interesting just for the beginning and the end of the lecture then the interface of the process consists of two events – begin and end.

The simplest possible process behavior stands for do nothing written as *STOP*. Whenever the behavior of a system reaches this process then deadlock occurred. Non trivial processes are written by means of prefixing operator which allows events to occur in sequence. So, when P is a process and a an event then $a \rightarrow P$ represent process which performs the event a and then behaves like the process P. Expressions of the type $P \rightarrow Q$ or $a \rightarrow b$ are not allowed. The prefix operator defines only the relation between events and processes (for example: a → P, a → b → P, etc.).

Except *STOP* another predefined process exists in CSP – *SKIP*. Like *STOP*, it does nothing but ends correctly. Therefore, the *SKIP* process indicates the correct termination of a process.

Utilizing predefined processes and the prefix operator only finite processes can be created. But often have to be specified processes that run forever. To achieve this goal recursion is included. For example, specification of a clock using an event *tick* describes the following process: $CLOCK = tick \rightarrow CLOCK$. The process *CLOCK* performs the event *tick* repeatedly.

Specified processes often don't just perform single sequence of events but may have alternative behavior caused by their environment, for example. So, if P and Q are processes and x and y are distinct events, then the process $x \rightarrow P \mid y \rightarrow Q$ performs either the event x and then behaves like process P or performs the event y and then behaves like process Q.

Modeled processes usually don't appear isolated but interfere with other process, for example with the process's environment. Mutual interaction between two or more processes means that these processes perform common events simultaneously. The alphabet of events specifies on which events the processes synchronize. For example, when describing the vending machine again, the new process representing the customer interacts with the machine. Example 1 describes these interacting processes.

Example 1:

$$MACHINE = coin \rightarrow (\, choc \rightarrow MACHINE \mid coffee \rightarrow MACHINE \,)$$
$$CUSTOMER = coin \rightarrow choc \rightarrow SKIP$$
$$SYSTEM = MACHINE \; {}_{A}||_{A} \; CUSTOMER$$
$$A = \{coin, choc, coffee\}$$

So far the utilized events were considered regardless of whether they represent inputs or outputs. However, separated notation for input and output may be useful for some cases. For this purpose a special event in the form $c.v$ is defined, where c stand for the communication channel name and v stand for the message value send through the channel. Each channel has a type which simply represents the set of events which can be transmitted among the channel. The notation $c.v$ is a general notation for communication channels. Therefore, to support sending and receiving of messages, two new operators are defined: process $c!v \rightarrow P$ sends a message v among the channel c and then behaves like P, process $c?x : T \rightarrow P(x)$ receives the message x of the type T and then behaves like $P(x)$. Until a message of the specified type appears on the input the receiving process waits.

The complete algebra of CSP provides much more notations but for this contribution purposes the presented subset fit the requirements. The complete algebra of Communicating Sequential Processes and its strict description is given in [2], while [3] presents more simplified version.

Failure Divergence Refinement (FDR) - FDR facilitate verification of many finite system properties and analysis of systems which fail the test. It stem from the Communicating Sequential Processes theory and utilizes refinement theory which provides huge range of correctness requirements including the absence of deadlock and livelock. FDR includes also requirements for general safety and liveliness properties.

4 Activity Diagrams from UML2

An activity specifies the coordination of executions of subordinate behaviors, using a control and data flow model. The subordinate behaviors coordinated by these models may be initiated because other behaviors in the model finish executing, because objects and data become available, or because events occurs external to the flow. The flow of execution is modeled as activity nodes connected by activity edges. A node can be the execution of a subordinate behavior, such as an arithmetic computation, a call to an operation, or manipulation of object contents. Activity nodes also include flow-of-control constructs, such as synchronization, decision, and concurrency control. Activities may form invocation hierarchies invoking other activities, ultimately resolving to individual actions. In an object-oriented model, activities are usually invoked indirectly as methods bound to operations that are directly invoked.

Activities may be applied to organizational modeling for business process engineering and workflow. In this context, events often originate from inside the system, such as the finishing of a task, but also from outside the system, such as a customer call.

Activities can also be used for information system modeling to specify system level processes. Activities may contain actions of various kinds:

- occurrences of primitive functions, such as arithmetic functions.
- invocations of behavior, such as activities.
- communication actions, such as sending of signals.
- manipulations of objects, such as reading or writing attributes or associations.

Actions have no further decomposition in the activity containing them. However, the execution of a single action may induce the execution of many other actions. For example, a call action invokes an operation which is implemented by an activity containing actions that execute before the call action completes.

In Activity diagrams, various graphical nodes can be included representing particular actions in the specified behavior. The most important nodes included also in this contribution are presented in the following table:

Name	Symbol	Description
Initial Node	●	An initial node is a starting point for invoking an activity. A control token is placed at the initial node when the activity starts. Tokens in an initial node are offered to all

Proceedings of the 5th WSEAS Int. Conf. on SIMULATION, MODELING AND OPTIMIZATION, Corfu, Greece, August 17-19, 2005 (pp541-546)

		outgoing edges. If an activity has more than one initial node, then invoking the activity starts multiple flows, one at each initial node.
Activity Final Node		A token reaching an activity final node aborts all flows in the containing activity, that is, the activity is terminated, and the token is destroyed. All tokens offered on the incoming edges are accepted.
Action Node	Event	An action is an executable activity node that is the fundamental unit of executable functionality in an activity, as opposed to control and data flow among actions. The execution of an action represents some transformation or processing in the modeled system, be it a computer system or otherwise. The completion of the execution of an action may enable the execution of a set of successor nodes and actions that take their inputs from the outputs of the action.
DataStore Node	<<DataStore>>	A data store keeps all tokens that enter it, copying them when they are chosen to move downstream. Incoming tokens containing a particular object replace any tokens in the object node containing that object.
Decision Node		A decision node is a control node that chooses between outgoing flows. It has one incoming edge and multiple outgoing activity edges. Guards of the outgoing edges are evaluated to determine which edge should be traversed. The order in which guards are evaluated is not defined.
Merge Node		A merge node is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows.
Fork / Join Node		A fork node is a control node that splits a flow into multiple concurrent flows. A join node is a control node that synchronizes multiple flows. Fork and join nodes are introduced to support parallelism in activities.
Send signal Node	signal	SendSignalAction is an action that creates a signal instance from its inputs, and transmits it to the target object, where it may cause the firing of a state machine transition or the execution of an activity. The requestor continues execution immediately.
Accept Event Node	signal	AcceptEventAction is an action that waits for the occurrence of an event meeting specified conditions.
Pre- / Post-condition	<<LocalPreconditions>> constraint <<LocalPostconditions>> constraint	Local preconditions and postconditions are constraints that must hold when the execution starts and completes, respectively. They hold only at the point in the flow that they are specified, not globally for other invocations of the behavior at other places in the flow or on other diagrams.

5 Interconnecting Activity Diagrams and CSP

The in previous section described elements from Activity diagrams represent the basic building items for system's behavior description utilizing UML2. This section denotes the relation between these

elements and the CSP algebra. The behavior of each of the presented diagram elements can be described utilizing the CSP algebra so that the semantics of the diagram element stay unchanged. This relevance between the elements and the CSP algebra is presented in following paragraphs:

Initial Node – the semantics of this node denotes the beginning of a process which performs particular activities. In CSP, such semantics correspond to the definition of a new process, i.e. PROC = …

Final Node – the final node represents the end of a specified behavior. This node is reached whenever the specified process successfully terminates its behavior, therefore, it corresponds to the predefined process SKIP from the CSP algebra.

Action Node – this node stands for the executable and visible part of the behavior. It corresponds to an event in the algebra of CSP.

DataStore Node – when a process behavior reaches this node, the input event to this node is copied into particular variable and it appears that the event never leaves the DataStore node. CSP does not contain directly relevant object which would correspond to the given semantics, but it can be simulated by utilizing communication channel. The behavior of datastore must satisfy following requirements: 1) the event on input has to be stored in the DataStore for independently long time period. 2) on the output of the DataStore should appear the stored event, even when reading the event from the DataStore repeatedly. 3) whenever an another event appears on the input, then the stored event is replaced by the new one.

All those presented requirements can be accomplished in CSP by utilizing two auxiliary processes operating above two communication channels. The first process reads an event from the input, stores it, and, when required, sends the event to output. The second process is activated only when accessing the stored event, and it cares for the repetitive storage of the event so it can be accessed again. The following CSP notation describes the desired behavior:

AUX = lock → left?event → unlock → AUX(event)
| right!event → right2!event → AUX
AUX2 = right2?val → lock → left!val → unlock → AUX2

The events lock and unlock stands for locking and unlocking the DataStore to prevent consistency by concurrent access. Such notation of DataStore can be utilized for storage of general types of events, therefore, when considering events in the form of natural numbers, the DataStore appears as shared variable of an integer type.

Decision Node – this node has multiple outputs, where that one is selected which's guarding action can be executed. In CSP this corresponds to the general choice operator where each of the operands agrees with the particular guarding items from the outputs.

Merge Node – on the contrary to the decision node, the merge node connects concurrently provided parts of specified processes into one single continuation. CSP does not provide such element but this behavior can be substituted by ending the particular concurrent processes with a single successor. This successor is a new process which defines the behavior of the single continuation from the specification.

Fork / Join Node – the fork node divides one input process into few particular processes performed concurrently, the join node connects the concurrent processes into one. On the contrary to merge note, the join node synchronizes all the input processes. Again, CSP has no relevant elements but they have to be substituted by a set of processes. When performing a fork, the input process must terminate his behavior and a few successor processed have to be started. Such behavior can be achieved by adding new event on the end of the input process which starts all the successor processes. This is performed by adding the same event as the first event in the behavior to all the successor processes. The join of processes is represented in the same way, except that all the input processes synchronize with each other on the added event. The union of particular processes into single one is then performed as described in the Merge node.

Send Signal Node – this action sends a signal. Because the send event action provided by CSP synchronizes with the receipt of the event is it necessary to simulate the required behavior by an auxiliary process. This process reads the event from the channel and stores it temporally so that the sending process can continue in its behavior.

Accept Event Node – again, same as in the previous case, this action corresponds to the receive event from communication channel action from the CSP.

Pre / Postconditions – those elements can constrain performing of an action in the specified behavior to proceed only when the specified constrain holds. This goal can be achieved in two different ways. The first way utilizes shared variables as described in the DataStore node and evaluating the given constrain manually, utilizing a special processes. The second way lies in utilizing control points and the keyword 'assert' in FDR. When using the special processes, only constrains on integer values can be checked. Those processes compare two integer values and perform the corresponding event denoting whether the checked variable is greater, equal or less than the checked value. The comparison is performed by iterative decrementing both values. CSP specification of the comparison processes is following:

COMPARE → compare.x.y → LOOP(x, y)

LOOP(x, y) → compare.0.0 → equal → COMPARE
| compare.x.0 → greater → COMPARE
| compare.0.y → lower → COMPARE
| loop → LOOP(x-1, y-1)

This denoted relevance between the activity diagrams and the CSP algebra provides the base idea for a developed tool, which process a specification of a system denoted utilizing UML activity diagrams and translates this specification into relevant CSP representation. That representation can then be verified using FDR. The user does not need to learn new specification techniques but utilizes the common ones although the system correctness can be verified.

6 Conclusions

The usage of computer based systems grows from year to year. Many of those systems impact out lives and a failure would lead to huge losses, extremely to human deaths. Therefore, these systems need to be reliable. A major field in increasing the system reliability lies in utilization of formal methods. They can extremely increase the system safety and can help detect errors in earlier stages of the software design process. Unfortunately, formal methods are mostly mathematically based methodologies which are not familiar to common system developers. Moreover, the project leaders omit the formal specification and verification phase in the software development process because they think that the invitation of such phase increases the amount of time and human resources spent on the project.

The main motivation for this contribution is to prevent such prejudices and to help system designers to invite formal methods in the development process. Because the main disadvantage of formal methods is the lack on user friendly interface, few goals in this field were specified. The main purpose of these goals lies in no more development of new formalisms, but utilizing the existing ones and applying them on common software design methodologies.

This contribution presents the relevance between the Activity Diagrams from UML2 and the algebra of Communicating Sequential Processes. The main nodes utilized in Activity Diagrams are presented, together with the corresponding notation in CSP. This relevance represents the kernel for a developed tool for system designers. The user of the developed tool specifies a system behavior utilizing UML2 in common way, the tool translates this specification into relevant CSP representation which can then be verified utilizing common verification tool.

Further development in this area will be oriented on extension of the utilized diagrams elements to comprehend the whole Activity Diagrams so that the system designers has no constrains in the specification phase.

7 Acknowledgement

Supported by the Grant Agency of the Czech Republic through the grant GACR 102/05/0723: A Framework for Formal Specifications and Prototyping of Information System's Network Applications.

References:

[1] Clarke, E.M.,jr., Grumberg, O., Peled, D.A.: Model checking, The MIT Press, London, 2000, ISBN 0-262-03270-8

[2] Hoare C.A.R.: Communicating sequential processes, Prentice-Hall 1985, ISBN 0-13-153271-8

[3] Schneider, Gay: Concurrent and real time systems, http://www.cs.rhbnc.ac.uk/books/concurrency/course/index.html, 2001

[4] UML 2.0 Superstructure Specification, http://www.omg.org, 2004

[5] Šculík František, Švéda Miroslav: Automatically Generated CSP Specifications, In: Proceedings of the IEEE TC-ECBS and IFIP WG10.1 Joint Workshop on Formal Specifications of Computer-Based Systems, 2003, Huntsville, AL, US, US, 2003, p. 41-47

[6] Muan Yong Ng, Michael J. Butler: Tool Support for Visualizing CSP in UML. ICFEM 2002: 287-298

[7] Charles Crichton, Jim Davies, Alessandra Cavarra, A Pattern for Concurrency in UML, Oxford University Computing Laboratory, England, December 2001

[8] Christie Bolton, Jim Davies, Activity Graphs and Processes, In W. Grieskamp, T. Santen and W. Stoddart, editors, Proceedings of IFM 2000. Springer, 2000

[9] K. Havelund, N. Shankar, Experiments in Theorem Proving and Model Checking, Formal Methods Europe FME '96, Springer-Verlag, Oxford, UK. March, 1996, Pages 662, 681

[10] Peter Henderson, Bob Walter, Behavioural analysis of Komponent-Based Systéme, In: Information and Software technology, 2001, vol. 43, No 3., pp 161-169