Journal of Theoretical and Applied Information Technology
20th December 2013. Vol. 58 No.2
© 2005 - 2013 JATIT & LLS. All rights reserved·

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

FLOW-BASED CONCEPTUAL REPRESENTATION OF PROBLEMS

SABAH AL-FEDAGHI
Assoc. Prof., Department of Computer Engineering, Kuwait University, Kuwait
E-mail: sabah@alfedaghi.com

ABSTRACT

Well-structured problem-solving involves a stage of (external) representation of the problem to provide a structure that serves as a shareable object of thought for studying the behavior of the underlying system. A dominant problem-solving methodology involves state-based representation with an initial state, a goal state, and a set of transactions. In addition, object-oriented methodology and Unified Modeling Language (UML) are increasingly utilized for drawing descriptions of a problem space. Petri nets attract designers (e.g., software) with its formal depictions modeling the behavior of a system. Nevertheless, each of these methodologies of problem representation has its own weaknesses, especially with regard to incorporating the features of understandability and simplicity. This paper proposes a different flow-based representation that has advantages for describing certain types of problems. The resultant description is characterized by uniform application of the basic structure of a flow system. The new methodology is demonstrated through sample toy problems, such as the problem of the dining professors.

Keywords: *Conceptual Model, Dinning Philosophers, Problem Representation, Problem Solving, State-Based Representation, System Behavior, UML*

1. INTRODUCTION

According to Hong [1], “Problem solving has been one of the dominant fields of research in the study of human information processing over the past three decades.” Gestalt psychologists have researched how to define a problem and develop a solution and claim that such a process involves “insight” and “restructuring”. Inability to conceptualize a problem can hinder solving it. Newell and Simon's 1972 "Problem Space Theory" [2] has greatly influenced development of problem-solving methodologies. They proposed that problem-solving involves a search in a *problem space* that has an initial state, a goal state, and a set of transactions. The solution is achieved by starting with the initial state and passing through to the goal state, moving from one state to the next while applying heuristics or algorithms that systematically check all potential states. Some problem spaces are so large that it is very difficult, or sometimes impossible, to represent the entire space; thus strategies or heuristics are used to move through them. *A well-structured problem* consists of a well-defined initial state, a known goal state, a constrained set of the logical state, and constraint parameters [3].

Well-structured problem-solving involves three stages: (external) *representation* of the problem, a search for a solution, and implementation of the solution [1]. According to Simon [4], “solving a problem simply means representing it so as to make the solution transparent.... the ease of solving a problem is almost completely determined by the way the problem is conceptualized and represented.… A well-chosen analogy or diagram can make all the difference when trying to communicate a difficult idea to someone, especially a non-expert in the field” [5].

> Experts are better problem solvers than novices for a number of reasons. The most important reason is that [experts] construct richer, more integrated mental *representations* of problems than do novices ... Experts are better able to classify problem types … because their *representations* integrate domain knowledge with problem types. [6] (Italics added)

A representation in this context includes the *underlying structure of possible solutions* [7]. The quality of this representation directly influences solving of the problem [2][8]. Problem representation consists of organizing and displaying the problem and the information required to solve it, in ways that are appropriate for the problem-

© 2005 - 2013 JATIT & LLS. All rights reserved·

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

solving process. This has been addressed in various ways. This paper focuses on conceptual diagrammatic representations that assist in understanding of the semantics of the problem, and the development of mental models necessary to create working solutions [9]. In particular, we target initially pedagogical applications of the results introduced in the paper, since "a good way of … teaching all kinds of CS topics is to use visualization and graphic animation in their various guises" [10].

"Problem representation" can serve a number of functions [11], including the following:
- Guiding further interpretation of information about the problem
- Providing insight into the structure of the problem
- Simulating the behavior of the system based on knowledge of the properties of the system
- Developing a possible solution

(External) representations also "provide a structure that can serve as a shareable object of thought… When someone externalizes a structure, they are communicating with themselves, as well as making it possible for others to share with them a common focus. An externalized structure can be shared as an object of thought" [12].

The representation is constructed on the basis of the problem statement, including its features and environment. A certain problem may have a number of different representations, with each being more advantageous for solving the problem than the others in some way. Representing the problem can be achieved "by constructing tabular, graphical, symbolic or verbal representations, and shifting between representational formats; and *formulating* hypotheses by identifying the relevant factors in the problem and their interrelationships; organizing and critically evaluating information" [13]. According to [14], "Experience shows that formalisms endowed with graphical descriptions are more accepted by cross-organization's stake-holders (not just designers and programmers)."

Finite state machines are typically used to represent a problem with a finite number of states and input and output signals. Many tools, e.g., microwave ovens, vending machines, and washing machines, are controlled by the rules of a finite state machine. Other methods for representing problems include dataflow diagrams and Petri nets. Recently, UML has emerged as a tool for creating standard representations of designs and implementations. It is not a methodology; rather, it is a diagrammatic representation that aids in the description and understanding of systems and offers abstract models of systems. Because of the extent of the topic of methodologies of representation in the field of problem-solving, we give UML a little more focus as a particular scheme in this context.

Object-oriented methodology and Unified Modeling Language (UML, [15]) are increasingly utilized in modeling systems (e.g., software development). After all, a system model is a conceptual representation that reflects the dynamic behavior of its components that replicate the system's conditions and activity.

UML has evolved through extension, especially in its dynamic representation capabilities. Formal semantics have been introduced into existing informal notations [16][17]; for example, translation algorithms are defined to provide given specification notations with fixed semantics (e.g., UML diagram to Petri nets). "The results are particular formalizations of some notations, which, even if well suited for some application domains, cannot easily be generalized" [18]. The next section reviews a sample toy problem and its representation in UML diagrams and Petri nets. This problem is then re-represented in our flow-based description, providing a study case that we offer the reader to contrast various representation methodologies and to demonstrate the unique features of our new diagrammatic technique.

2. THE DINING PHILOSOPHERS PROBLEM

The Dining Philosophers Problem was posed and solved by Edsger Dijkstra [19] and is often used in studying concurrency, deadlock, and synchronization issues. Five philosophers sit around a table. A chopstick is placed between each pair of adjacent philosophers. They spend their time alternately thinking and eating. A philosopher can eat only when he has both left and right sticks. A philosopher can use a stick only if it's not being used by another philosopher. After he finishes eating, he needs to put down both sticks so they become available to others. The problem is to design a solution such that each philosopher won't starve. A possible solution is as follows.

For each philosopher:
Repeat
pick up left stick
pick up right stick
eat
put the left and right sticks down

© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

This solution permits a deadlock [19]. Another solution could incorporate a rule that the philosophers put down a stick after waiting a certain time and then waiting a little longer before trying again. This scheme eliminates the possibility of deadlock but still suffers from the problem of starvation. Dijkstra [19] solved the problem by assigning a partial order to the resources (sticks).

Many solutions have been presented for this problem, including one in which the philosophers and chopsticks are conceptualized as separate classes of objects that communicate via messages. Here, we are interested in the methodology of describing the problem. For example, [20] assumes a head waiter whose task is to receive messages from the philosophers, assign chopsticks, and serve as a play-by-play announcer. In this solution the waiter acts as a "semaphore" that controls access to a common resource by multiple processes in a parallel programming environment.

Another solution is that philosophers can eat if neither of their neighbors are eating, whereas philosophers who cannot get the second stick must put down the first stick before trying again. A single mutual exclusion lock is used with the decision procedures that can change the states of the philosophers. To guarantee that no philosopher starves, one could keep track of the number of times a hungry philosopher cannot eat when his neighbors put down their sticks, and a decision procedure for picking up sticks could be augmented to require that none of the neighbors are starving. A solution by Chandy and Misra [21] permits an arbitrary number of philosophers with an arbitrary number of resources and specifies that "the philosophers do not speak to each other".

According to Samek [22], the solution to the problem of the Dining Philosophers is a "need event-driven system": "In active object systems, the generic design strategy for handling such shared resources is to encapsulate them inside a dedicated active object and to let that object manage the shared resources for the rest of the system" [22]. The problem can be represented by drawing a UML sequence diagram (Figure 1) for the main scenarios (main use cases). Figure 1 shows the most representative event exchanges among any two adjacent philosophers and the Table active objects.

> Sequence diagrams … help you discover events exchanged among active objects. The choice of signals and event parameters is perhaps the most important design decision in any event-driven system. The events affect the other main application components: events and state machines of the active objects. [22]

Figure 2 shows the state machines associated with the Philosopher active object in which the life cycle consists of the states "thinking", "hungry", and "eating". "This state machine generates the HUNGRY event on entry to the "hungry" state and the DONE event on exit from the "eating" state because this exactly reflects the semantics of these events… This actually is the general guideline in state machine design" [22].

While this presents an unfair description of the solution to the problem, it is sufficient for our purpose, which is focused on the *representation* of the problem, not on the scheme of the solution. This paper will suggest an alternative conceptual description based on the notion of flow.

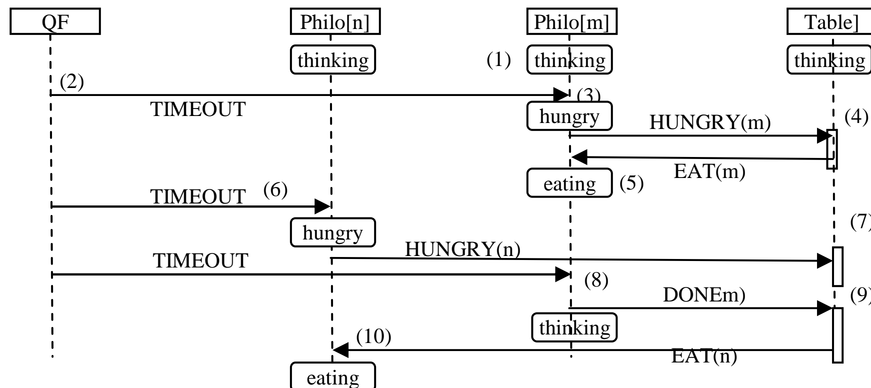

Figure 1. The Sequence Diagram Of The DPP Application (Partial, From [22])

© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

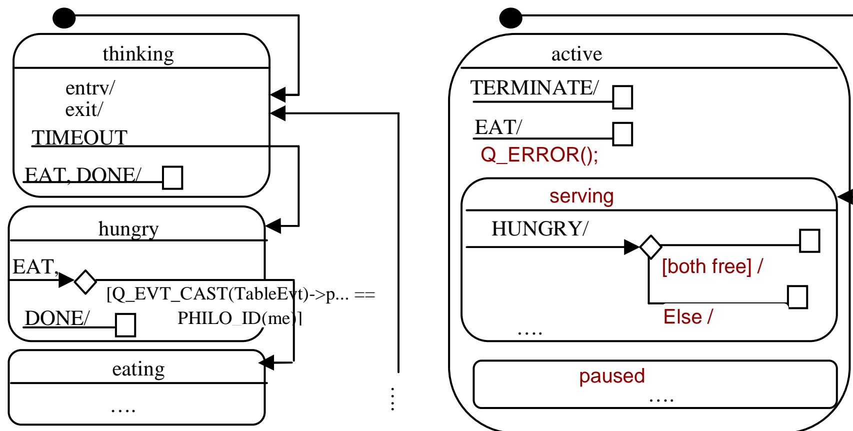

Figure 2. State Machines Associated With The Philosopher Active Object And Table Active Object (Partial, From [22])

Examining these diagrams, one might wonder whether such representations are suitable as a first step in approaching the solution, which is to draw a conceptual description of the behavior needed to arrive at the solution. A conceptual description ought to be completely, to use Jackson's terminology [23], independent from structuring the world domain in the machine domain. The process described above jumps from the English description to specification of sequence and state diagrams. The result seems to indicate a gap in communicating the nature of the problem and its proposed solution. The fragmented conceptual picture of sequence diagram and state diagram, patched with English and hindered by programming details, creates a mosaic of shifting focus for the problem solver and learner. Additionally, "The Unified Modeling Language has been shown to be complex and difficult to learn. The difficulty of learning to build the individual diagrams in the UML, however, has received scant attention" [24].

Another known framework for describing behavioral aspects in different kinds of problems is the Petri net specification. Petri nets differentiate between states and activities with graphical representation in addition to formal description. Figure 3 shows a version of the Petri net representation of the dining philosophers problem.

> Philosopher Pi may be in one of the two states, either eating or thinking, corresponding respectively to (presence of a token in) the places Pi_E and Pi_T. Each fork is modeled by a corresponding place, where the presence of a token indicates the availability of the fork. When philosopher's state changes from thinking to eating (resp. eating to thinking), the two forks on its left and right become no more available (resp. available again). Initially, all philosophers are thinking and thus all forks are available. [14]

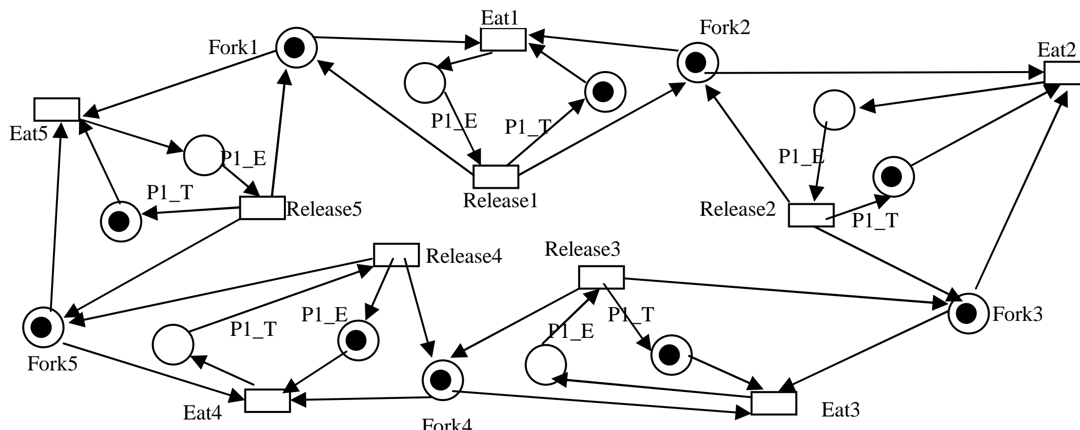

Figure 3. The Dining Philosopher Problem As A Petri Net (From [14]).

© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Applying "firing rules" to the initial marking of Figure 3 results in new markings. There are many variants of Petri-net extensions for different reasons (e.g., [25]–[29]). Zafar [14], for example, proposes algebraic Petri nets to "support the construction of concise, but nevertheless comprehensible and transparent models of real-world systems." Nevertheless, according to Zafar [14],

> Formalisms such as high-level Petri nets still remain hardly understandable and accepted by (cross) organization stakeholders (e.g. managers, users, customers and even programmers), we are going to promote the practicability and the wide-usability through the early adoption of semi-formal diagrammatical and standardised artifacts both for structural and behavioral features in service-driven applications. More precisely, all structural features of service-driven business applications are first described using stereo-typed *UML 2.0* use-cases and class-diagrams. Behavioral aspects are captured through *event-driven business rules*, which are inherently understandable, evolving and process-independent. *Only after such widely acceptable and accessible semi-formal descriptions*, of any service-driven application at hand, we then forward a smooth and semi-automatic shifting towards the proposed rigorous service-driven Petri Nets formalism. [14] (Italics added)

Hence, such factors as the huge investment in development of standard ULM, or the available formalism of Petri nets, ought not to discourage new ventures to propose other models with the same objectives. Accordingly, the contribution of this paper is to propose such a new model that can be incorporated into current methodologies.

In preparation for recasting the representation of the dining philosophers problem in terms of our methodology, and to make this paper self-contained, the next section briefly reviews published materials describing the model that will be used as the basis for a description [30–33]. This paper applies this model in the area of problem representation in the context of problem-solving.

3. FLOWTHING MODEL

To make this paper self-contained, the materials in this section are summarized from a series of papers that have applied the model in several application areas [30]-[33].

The Flowthing Model (FM) was inspired by the many types of flows that exist in diverse fields, such as, for example, supply chain flow, money flow, and data flow in communication models. This model is a diagrammatic schema that uses *flowthings* to represent a range of items that can be data, information, objects, or signals. FM also provides the modeler the freedom to *draw* the system using *flowsystems* that include six stages, as follows:

- *Arrive:* A flowthing reaches a new flowsystem (e.g., a buffer in a router)
- *Accepted*: A flowthing is permitted to enter the system (e.g., no wrong address for a delivery); if arriving flowthings are also accepted, Arrive and Accept can be combined as a *Received* stage.
- *Processed (changed):* The flowthing goes through some kind of transformation that changes its form but not its identity (e.g., compressed, colored, etc.)
- *Released:* A flowthing is marked as ready to be transferred (e.g., airline passengers waiting to board)
- *Created:* A *new* flowthing is born (created) in the system (a data mining program generates the conclusion *Application is rejected* for input data)
- *Transferred:* The flowthing is transported somewhere outside the flowsystem (e.g., packets reaching ports in a router, but still not in the arrival buffer)

These stages are mutually exclusive, i.e., a flowthing in the process stage cannot be in the created stage or the released stage at the same time. Figure 4 shows the structure of a flowsystem. A flowthing is a thing that has the capability of being created, released, transferred, arrived, accepted, or processed while flowing within and between systems. *A flowsystem* depicts the internal flows of a system with the six stages and transactions among them. FM also uses the following notions:

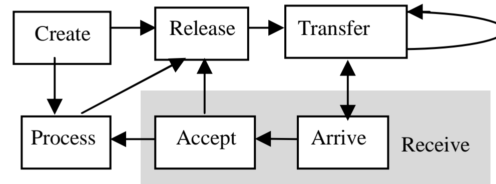

Figure 4. Flowsystem

- *Spheres and subspheres*: These are the environments of the flowthing, such as a company and the departments within it, an instrument, a computer, an embedded system, a

Journal of Theoretical and Applied Information Technology
20th December 2013. Vol. 58 No.2
© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

component, and so forth. A sphere can have multiple *flowsystems* in its construction if needed.
- *Triggering*: Triggering is a transformation (denoted in FM diagrams by a dashed arrow) from one flow to another, e.g., flow of electricity triggers the flow of air. If a sphere has one flowsystem, then the two flows can be represented by one box.

A flowsystem need not include all the stages; for example, an archiving system might use only the stages *Arrive, Accept,* and *Release*. Multiple systems captured by FM can interact with each other by triggering events related to one another in their spheres and stages.

It may be argued that data can be in a *stored* state, which is not included as a stage of a flowsystem*;* however, *stored* is not a primary state, because data can be stored after being created, hence it is *stored created data*, or it is stored after being processed, hence it is *stored processed data*, and so on. Because current models of software and hardware do not differentiate between these states of stored data, we will assume flowsystems with unified storage.

In addition to the fundamental characteristic of flow in FM, the following types of possible operations exist in different stages:

1. *Copying*: Copy is an operation such that *flowthing* f => f. That is, it is possible to copy f to produce another flowthing f in a system S. In this case, S is said to be S *with copying* feature, or, in short, *Copy S*. For example, any informational flowsystem can be copy S, while physical flowsystems are non-copying S. Notice that in copy S, stored f may have its copy in a non-stored state. It is possible that copying is allowed in certain stages and not in others.

2. *Erasure*: Erasure is an operation such that *flowthing* f => e, where e denotes the *empty flowthing*. That is, it is possible to erase a flowthing in S. In this case, S is said to be S *with erasure* feature, or, in short, *erasure S*. Erasure can be used for a single instance, all instances in a stage, or all instances in S.

3. *Canceling*: *Anti-flowthing* f^- (f with superscript –) is a flowthing such that (f^- + f) => e, where e denotes the empty flowthing, and + denotes the presence of f^- and f.

It is possible that the anti-flowthing f^- is declared in a stage or a flowsystem. If flowthing f triggers the flow of flowthing g, then the anti-flowthing f^- triggers anti-flowthing g^-.
An example of the use of these FM features is erasure of a flow, as in the case of a customer who orders a product, then cancels the order. This may require cancellation of several flows in different spheres triggered by the original order.

Formally, FM can be specified as FM = $\{S_i(\{F_j\}, T_l), \{(F_{ij}, F_{ij})\}, 1 \le i \le n, 1 \le j \le m, 1 \le l \le t\}$
That is, FM is a set of spheres S_1, ...S_n, each with its own flowsystems F_{ij},... F_{im}. T is a type of flowthing T_1,..., T_t. Also, F is a graph with vertices V that is a (possibly proper) subset, {Arrive*, Accept*, Process*, Create*, Release*, Transfer*}, where the asterisks indicate secondary stages.

Example: Consider the "Vacuum World" toy problem [34]. It is noted that "The problem-solving approach has been applied to a vast array of task environments. They tend not to have a single agreed-upon description, so we will do our best describing the formulations used" [34]. The problem is formulated as follows [34]:
- **States:** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt.
- **Initial State:** Any state.
- **Actions:** Each state has just three actions: *Left, Right,* and *Suck*
- **Transition:** The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Sucking* in a clean square have no effect.
- **Goal Test:** This checks whether all the squares are clean.
- **Path Cost:** Each step costs 1, so the path cost is the number of steps in the path.

Figure 5 shows the state space for Vacuum World.

Figure 6 shows the corresponding FM representation of the same problem. Assume that the cleaning machine starts in location 0, cleans the dirt (if there is any), then flows to location 1 to do the same thing, then stops. A simple pseudo code may be presented as follows:

If location (i), then clean it;
Move to location (Mod(i+1));
If location (i), then clean it;
stop;

© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

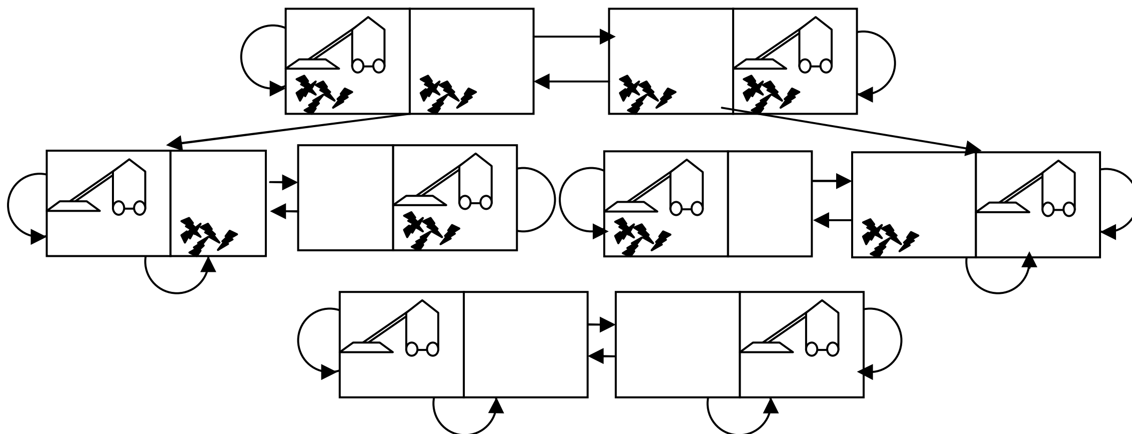

Figure 5. The State Space For The Vacuum World (From [34]).

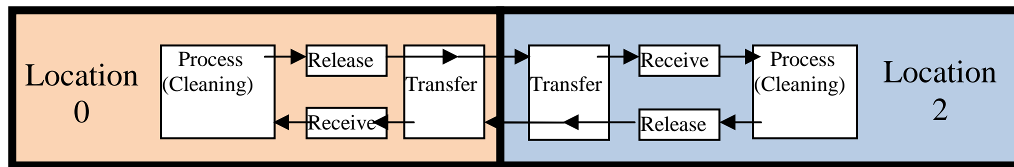

Figure 6. FM-Based Representation Of The Vacuum Cleaning Problem

Note the difference in the size of representations in Figure 6 in comparison with Figure 5, where n locations has n * 2^2 states; thus, for 4 locations there are 16 states.

Figure 7 shows the FM representation of 4 locations. For example, if the machine is in location 0, it can move to location 1 or 3. Writing a solution for this case is not difficult.

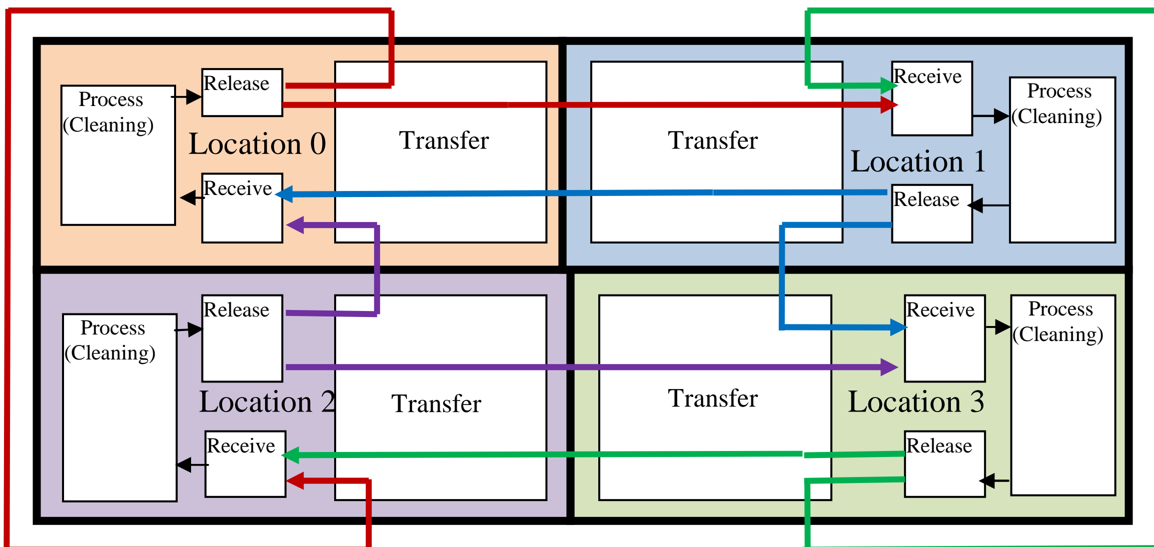

Figure 7. FM-Based Representation Of The Vacuum Cleaning Problem With 4 Locations

© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

4. DINING PHILOSOPHERS PROBLEM REVISITED

Returning to the Dining Philosophers problem, Figure 8 shows the FM-based representation. Instead of sequence and statechart diagrams complemented with annotations and descriptions that try to present a glued-together single conceptual account, FM provides a uniform description in a single framework.

Starting at circle 1 in Figure 8, a stick is released from the table. This means that a philosopher grabs the stick, and from the table's point of view, it is free to be used (3). When the philosopher actually moves the stick to his/her mouth, the stick is in the transfer state (circle 2). In this condition, scrutiny of the event occurs: is the right-hand stick also being transferred (4)? If not, the stick is marked as "not used" (5), and the philosopher is triggered (6) to the waiting state. In the waiting state, and after a fixed time (8), the philosopher tries (triggers – circle 9) to release (10) the stick again.

If the right stick is also being transferred, then the stick flows to the philosopher (11) to be received and used in eating. When he or she is finished eating (12), the philosopher releases the left stick (13), triggering him/her to think (14). Also, when the stick is received on the table (15), this triggers (16) a change in the state of the stick to "Not used". Identical events occur with respect to the right-hand stick (lower part of the figure).

One event can occur which we could not find in the sources about this problem, and that is when a philosopher releases one stick while not releasing the other (holding it in hand). It is possible to force release and transfer of both sticks simultaneously using a similar test as when forcing the two sticks to move simultaneously from the table (circle 4, and corresponding testing in the right stick flowsystem). In our case, we permit this situation; hence, the philosopher goes to the state of thinking.

In the state of thinking, after a fixed time (17), the philosopher tries (triggers – circle 18) to release (1) the stick again. Similar events occur with the right stick, starting from release.

Figure 8 can be used in searching for a solution to the Dining Philosophers problem. The search can start at any position of the philosophers in the diagram. Currently known strategies can be applied to the FM representation. Figure 9 shows a sample simple search pseudo code for simulating the events in the problem representation. It is assumed that the philosophers start in the thinking state and that they finish thinking one after another. The simulation can be ended in several ways, such as finding a cycle (every philosopher thinks, eats at least once) where no deadlock or starvation occurs.

However, we will not go into the details of a certain solution, since the objective of this paper is to demonstrate a new representation that can be used only for explaining the problem and its related features such as deadlock and concurrency.

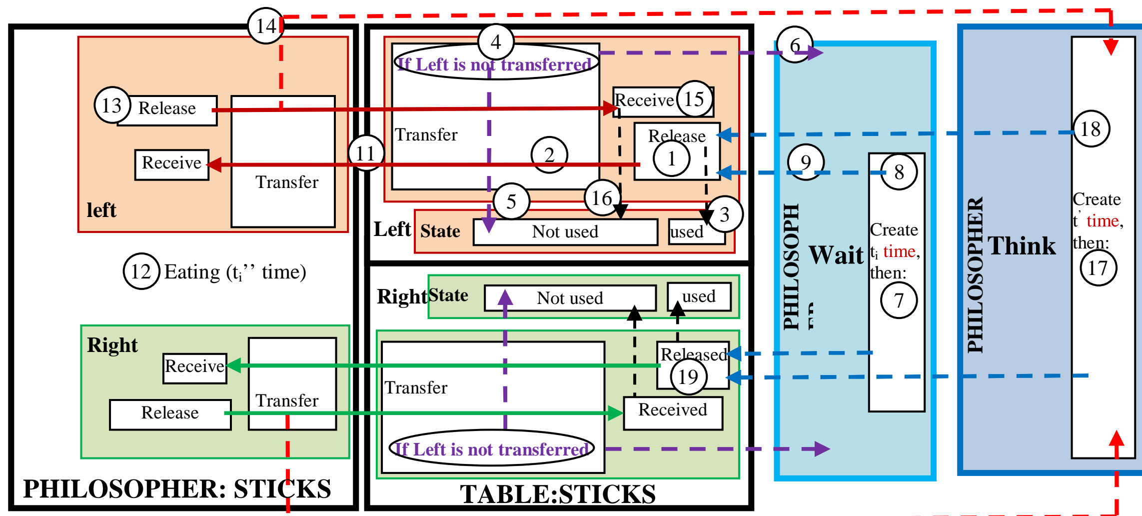

Figure 8. The FM representation of the Dining Philosophers problem

© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

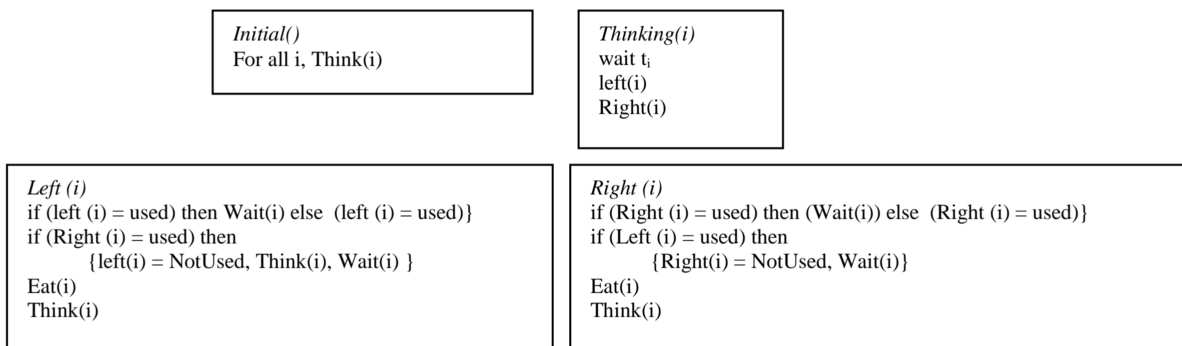

Figure 9. Possible Simple Pseudo Search For Solution To The Dining Philosophers Problem

5. CONCLUSION

Methodologies of representation used in problem-solving include state, UML, and Petri-net diagrams. Each has its own advantages and weaknesses, especially in regard to the features of understandability and simplicity. This paper has proposed a different flow-based representation that is based on the notion of flow and characterized by uniform application of the basic structure of a flow system. The new methodology is demonstrated through sample toy problems.

We are currently exploring areas of application for such a methodology of representation, as in the case of presenting it as a first phase in describing a problem to students, instead of, say, UML diagrams. It is possible that FM can be utilized in actual searches for solutions.

REFERENCES:

[1] Hong NS. (1998). *The relationship between well-structured and ill-structured problem solving in multimedia simulation*. PhD thesis, Pennsylvania State University. http://www.cet.edu/pdf/structure.pdf

[2] Newell A., & Simon HA. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice Hall.

[3] Greeno J (1978). Natures of Problem-Solving Ability. *Handbook of learning and cognitive processes* (pp. 239-270). Hillsdale, NJ: Lawrence Erlbaum.

[4] Simon HA. (1981). Studying human intelligence by creating artificial intelligence. *American Scientist* 1981; 69(3): 300-309.

[5] Epistemics. (2003). *Knowledge Models*. Blog. Last modified: 20 November 2003. Accessed October, 2012. http://www.epistemics.co.uk/Notes/90-0-0.htm

[6] Jonassen DH. (2005). Tools for representing problems and the knowledge required to solve them. *Knowledge and Information Visualization*. 82-94.

[7] Voss JF., Lawrence JA., & Engle RA. (1991). From representation to decision: An analysis of problem solving in international relations. In R. J. Sternberg RJ, Frensh PA (Eds.), *Complex problem solving*, pp. 119-157. Hilldale, NJ: Lawrence Erlbaum.

[8] Hayes JR. (1981). *The complete problem solver*. Philadelphia: The Franklin Institute Press.

[9] Miller JA. (2004). *Promoting computer literacy through programming Python*. PhD. dissertation, University of Michigan. http://www.python.org/files/miller-dissertation.pdf

[10] Gal-Ezer J., & Harel D. (1998). What (else) should CS educators know? *Communications of the ACM*; 41(9).

[11] Savelsbergh ER., deJong T., & Ferguson-Hessler, MGM. (1998). Competence-related differences in problem representations. In M. van Sommeren M, Reimann P, deJong T, Boshuizen H (Eds.), *The role of multiple representations in learning and problem solving*, pp. 262-282. Amsterdam: Elsevier.

[12] Kirsh D. (2010). Thinking with external representations. *AI & Soc*; 25:441–454. DOI 10.1007/s00146-010-0272-8.

[13] PISA (2012). *PISA 2012 Field trial problem solving* framework. http://www.oecd.org/pisa/pisaproducts/46962005.pdf

[14] Zafar B. (2008). *Conceptual modelling of adaptive Web services based on high-level Petri nets*. PhD thesis, De Montfort University. https://www.dora.dmu.ac.uk/bitstream/handle/2086/2407/thesis44.pdf?sequence=1

© 2005 - 2013 JATIT & LLS. All rights reserved·

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[15] Fowler M., & Scott K. (1997). *UML distilled: Applying the Standard Object Modeling Language*. Reading, MA: Addison-Wesley.

[16] Bates BW., Bruel JM., & France RB, Larrondo-Petrie MM. (1996). Guidelines for formalizing fusion object-oriented analysis methods. *In Conference on Advanced Information Systems Engineering (CAiSE) 96*, pp. 222–233.

[17] Wang EY., Richter HA., & Cheng BHC. (1997). Formalizing and integrating the Dynamic Model within OMT. *In Proceedings of the 19th International Conference on Software Engineering*, pp. 45–55. ACM Press, May.

[18] Baresi L., & Pezz M. (2001). On formalizing UML with high-level Petri nets, In: Agha G, et al. (Eds.), *Concurrent OOP and PN*, LNCS, pp. 276–304. Berlin: Springer-Verlag.

[19] Dijkstra EW. (1971). Hierarchical ordering of sequential processes. *Acta Informatica* 1, 115-138. http://www.cs.utexas.edu/~EWD/ewd03xx/EWD310.PDF

[20] Feldman MB. (1992). The portable dining philosophers: a movable feast of concurrency and software engineering. *Proc. 23rd ACM-SIGCSE Technical Symposium on Computer Science Education*, Kansas City, MO, March. http://www.seas.gwu.edu/~mfeldman/papers/portable-diners.html

[21] Chandy KM., & Misra J. (1984) The Drinking Philosophers problem. *ACM Trans. on Programming Languages and Systems* G:G32-646.

[22] Samek M (Quantum Leaps, LLC). (2012). Application note Dining Philosophers Problem (DPP) example. Document Revision D August. http://www.google.com.kw/url?sa=t&rct=j&q=%22application%20note%20dining%20philosophers%20problem%20(dpp)%20example%22&source=web&cd=1&cad=rja&sqi=2&ved=0CB0QFjAA&url=http%3A%2F%2Fwww.state-machine.com%2Fresources%2FAN_DPP.pdf&ei=ja5dUKHBGIi40QW2xIFw&usg=AFQjCNEmxvIJroeOeznmXNJfDrIyyYbaRA

[23] Jackson M. (1997). The meaning of requirements. *Annals of Software Engineering*; 3(1):5–21.

[24] Vander Meer D. (2009). Applying learner-centered design principles to UML sequence diagrams. *Database Management*; 20(1).

[25] Bernard B., & Michel D. (1991). Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Software Eng*: 259-273.

[26] Lakos C. (1995). From coloured Petri nets to object Petri nets. In *Proc. of 16th Application and Theory of Petri Nets*, Lecture Notes in Computer Science; 935: 278–287.

[27] Lakos C. (1996). The consistent use of names and polymorphism in the definition of objects Petri nets. In *Proc. of 17th Application and Theory of Petri Nets*, Lecture Notes in Computer Science; 1091: 380–399.

[28] Sibertin C. (1994). Communicative and cooperative nets. In *Proc. of the 15th International Conference on the Application and Theory of Petri Nets*, Lecture Notes in Computer Science; 815.

[29] Jensen K. (1992). Coloured Petri nets: basic concepts, analysis methods and practical use. Volume 1: Basic Concepts. *EATCS Monographs in Computer Science*, 26.

[30] Al-Fedaghi S. (2010). System-based Approach to Software Vulnerability, *The IEEE Symposium on Privacy and Security Applications (PSA-10)*, Minneapolis, USA.

[31] Al-Fedaghi S. (2009). Interpretation of Information Processing Regulations, *Journal of Software Engineering & Applications*, Vol. 2 No. 2, pp. 67-76.

[32] Al-Fedaghi, S. (2012). Conceptual Framework for Recursion in Computer Programming, *Journal of Theoretical and Applied Information Technology*, Vol. 46 No. 2.

[33] Al-Fedaghi S., & Al-Saqa A. (2013). Toward A Conceptual Base for Protocol Engineering, *Journal of Theoretical and Applied Information Technology*, Vol. 51, No. 2.

[34] Centurion. (2012). Well-defined problems and solutions: Vacuum World. Last Modified 5 May. http://centurion2.com/AIHomework/AI110/ai110.php

[35] Jumaat, S.A. , Musirin, I., Othman, M. M., & Moklis, H. (2012). Computational Intelligence Based Technique In Multiple Facts Devices Installation For Power System Security, *Journal of Theoretical and Applied Information Technology*. pp 0537 - 0549 Vol. 46. No. 2.

[36] Guo, C. (2013). Design and Implementation of a Multimedia Database Application system, *Journal of Theoretical and Applied Information Technology*. Vol. 47. No. 3 – January.