Using Transition Systems to Formalize a Pattern for Time Dependable Applications

Edward Hermann Haeusler and Marcus Fontoura
Computer Science Department, Pontifical Catholic University of Rio de Janeiro
Rua Marquês de São Vicente, 225, 22453-900 Rio de Janeiro, Brazil
{hermann, mafe}@inf.puc-rio.br

Abstract

This paper presents an example of the use of transition systems to formalize a pattern for soft real-time systems. The importance of this relies on the fact that this formalization can be used as a guide to a mapping from projects to transition systems based model-checkers. Since model-checking is a technique based on finite transition systems, this approach, currently, can only be applied to applications that do not modify dynamically the amount of interacting objects. Hence, the applicability of the pattern is limited to statically configurable systems.

The pattern, namely MULTI-AUTOMATA, is presented through a variation of the GoF form. A high-level specification language used to instantiate the pattern is also presented. The high-level language respects the pattern semantics and is used to help the pattern utilization.

1. The multi-automata pattern

This section presents the MULTI-AUTOMATA pattern through a variation of the GoF pattern form [6]. The pattern utilization is described through the use of a high-level specification language. The implementation section discusses the C++ code generation from the high-level specification [7].

INTENT

Promote synchronous behavior among several interacting and concurrent elements in a soft[1] real-time environment. MULTI-AUTOMATA also allows asynchronous message passing between the elements.

MOTIVATION

Consider a multimedia presentation environment where several different components interact, each one presenting a different multimedia element like text, audio, video, and still images. Each of these components should have its own execution thread. Also, synchronization mechanisms among the components should be specified to control the presentation.

[1] This paper considers soft real time systems as not concerning concrete time specifications, but relative synchronization constrains among the interacting elements.

Figure 1 illustrates a scenario where a presentation is composed of a book (text structured into chapters), videos, and still images. Each book chapter has its own video and set of pictures. The user can browse the presentation elements in any order, but when the current chapter is changed in one of the elements, the other two elements must also change.

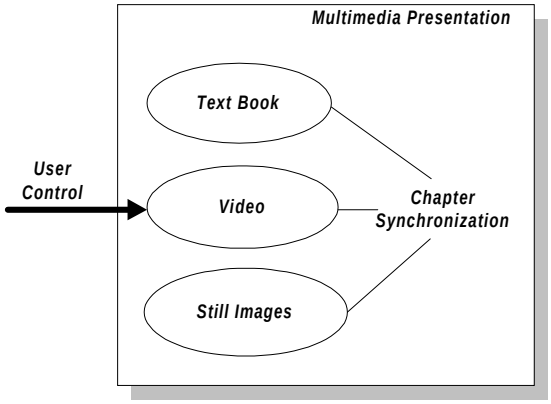

Figure 1. Multimedia presentation example

This means that if the user current focus is on the video element, and the video being presented is the one related to chapter 2, when she decides to see chapter 3 video, the book and the still images must also change.

In a general case, possibly new multimedia elements would be added to the system and more complex synchronization constrains could be needed. The MULTI-AUTOMATA pattern provides a high level language to specify synchronization constrains among interacting objects. The language has constructors for specifying state transition diagrams for each class, synchronization constrains, and asynchronous message passing.

Taking a deeper look into the multimedia presentation example, one possible specification of the application is the one presented in Figure 2. Four state diagrams are presented: one for each interacting element and one for the user. The language constructor "sync" specifies which transitions should happen at the same time. The user has only one state, which is "Watching". The asynchronous message "set(chapter ← chapter1)" is sent to the object aBook, modeled by the Book state diagram, in the beginning of the system execution.

The presentation starts when aBook handles the message. The pattern guarantees that the three presentation objects will start showing their content at the same time because there is a synchronism specified for the transitions NextChapter, NextVideo, and NextChapterImages. The same holds for the finishing transitions, assuring that the three objects always show the content related to the same chapter.

The design structure presented in Figure 3 is a simplified utilization of the MULTI-AUTOMATA pattern for this example. Each one of the presentation objects is a sub-class of the abstract class InteractingElement and has a message queue (queue) to handle asynchronous messages. The behavior of these objects is specified by the ShowChapter method: they have to check the message queue, handle the first message (if any), and then show the content of the current chapter. The static variable chapter stores the value of the current chapter.

The MultimediaPresentation class aggregates all the interacting objects. Note that the configuration of how many interacting objects of each class are presented has to be statically defined. This is a limitation of the pattern, which can be applied only to statically configurable systems.

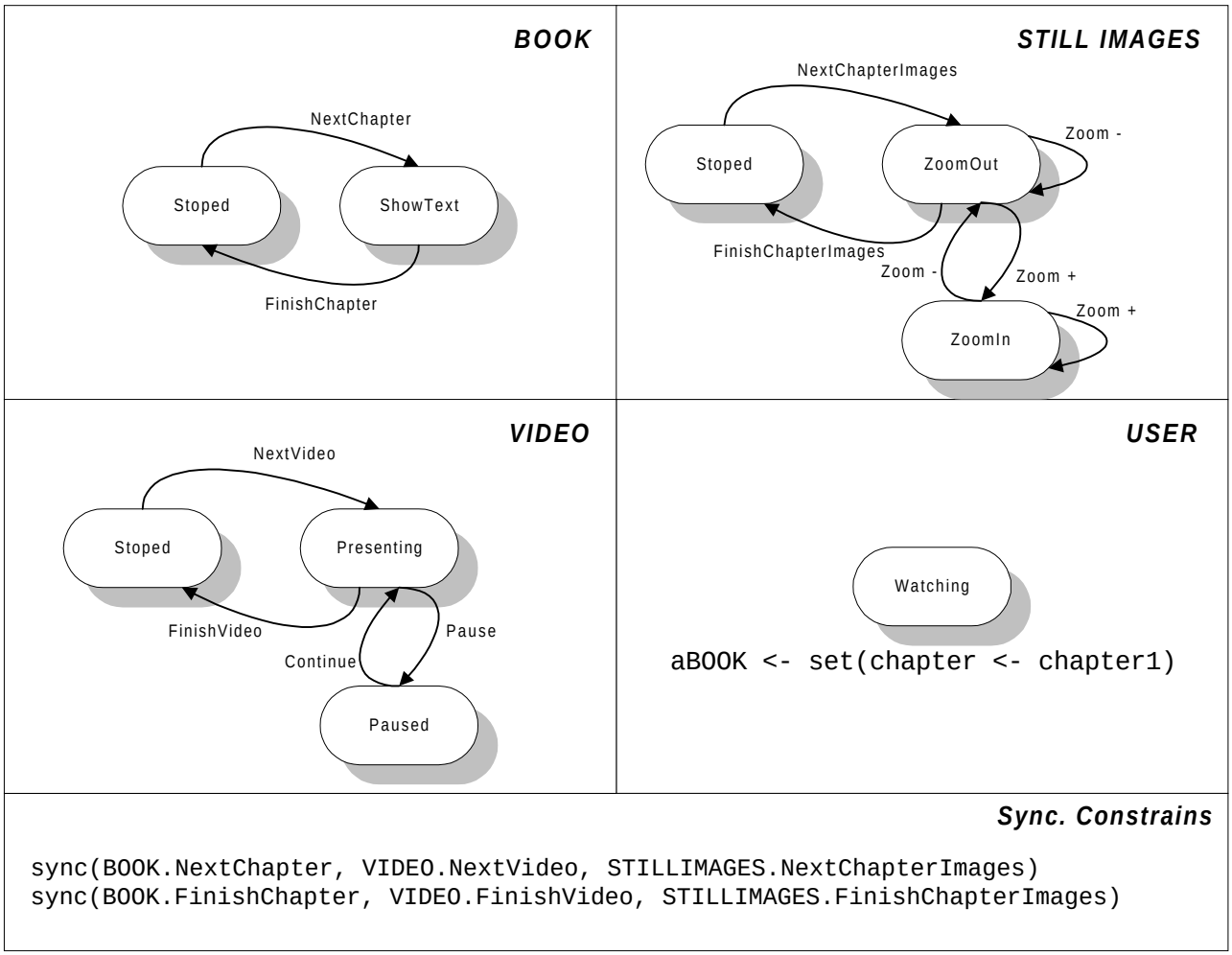

Figure 2. Modeling the system with a high level language

All the synchronization constrains have to be treated by the MultimediaPresentation class, more specifically, by the ShowPresentation method. The implementation of the synchronization mechanisms uses low-level operating systems classes and will not be shown here.

APPLICABILITY

Apply MULTI-AUTOMATA when all of the following are true:

- Several elements interact in a soft real time environment;
- The elements have synchronization constrains;
- They may communicate via asynchronous message passing;
- The system is statically configurable, that is, the number of objects of each interacting element class can be predefined.

STRUCTURE

Figure 4 shows the design model of the MULTI-AUTOMATA pattern, through the use of an OMT-like class diagram.

PARTICIPANTS

- MessageQueue: is used for storing the asynchronous messages. Each interacting element has its own message queue.
- InteractingElement: provides a default abstract behavior for each interacting object.
- Project (MultimediaPresentation): the only behavior specified in a project is the (static) configuration of the aggregate objects and the synchronization control. There is no loss of generality with this approach since any additional behavior can be refactored to an extra interacting element object. Note that projects can aggregate interacting elements and projects (sub-projects) as well.

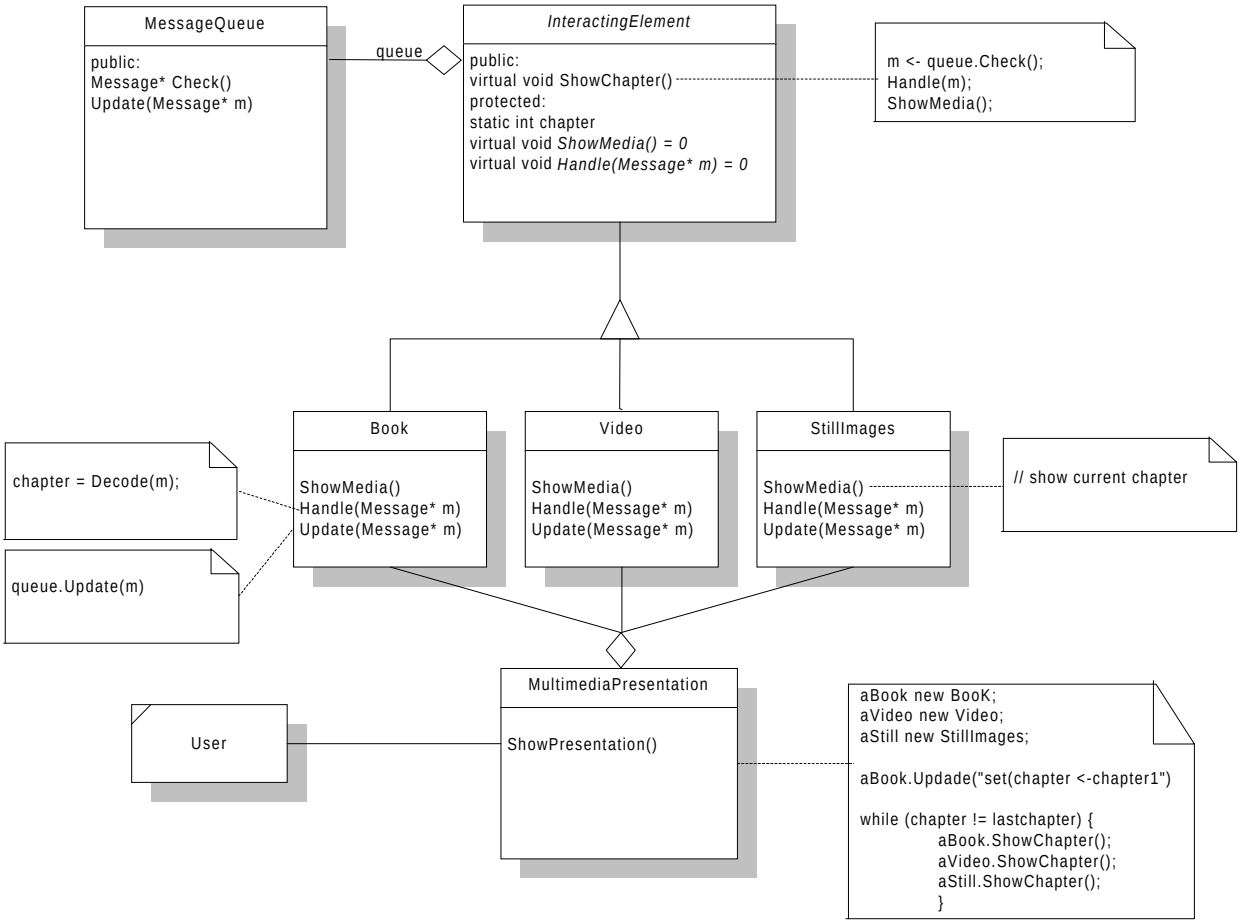

Figure 3. Applying MULTI-AUTOMATA

IMPLEMENTATION

In the pattern current implementation, user intentions are described in a high-level language (Figure 2) and a transformational system is used to generate the C++ code that implements the pattern from this high-level specification. In [2] a general approach for instantiating object oriented frameworks [8] from domain specific languages is presented.

Currently, the implementation of synchronization mechanisms is based on Microsoft Windows 95 low-level classes.

KNOWN USES

ARTS-III is an environment for supporting object-oriented software development based on formal methods [7]. The MULTI-AUTOMATA pattern, together with its high level specification language, was used as part of the project for building a framework for PABXs that is being used by industry (Siemens Telecommunications - Brazil).

The applicability of MULTI-AUTOMATA to other domains will be tested as part of the project. The multimedia presentation domain is the next to be investigated.

RELATED PATTERNS

The MULTI-AUTOMATA mechanism for asynchronous message passing can be implemented as a variation of the OBSERVER design pattern [6, 9] notification mechanism. The MULTICAST pattern [9] presents a design solution for event communication that could also be used.

The objects default behavior can be specified through the use of the TEMPLATE METHOD design pattern [6]. The STATE design pattern [6] can be used to model the MULTI-AUTOMATA state diagrams. Since in the current implementation of MULTI-AUTOMATA its design structure is generated from the specification language, there was no need to apply STATE.

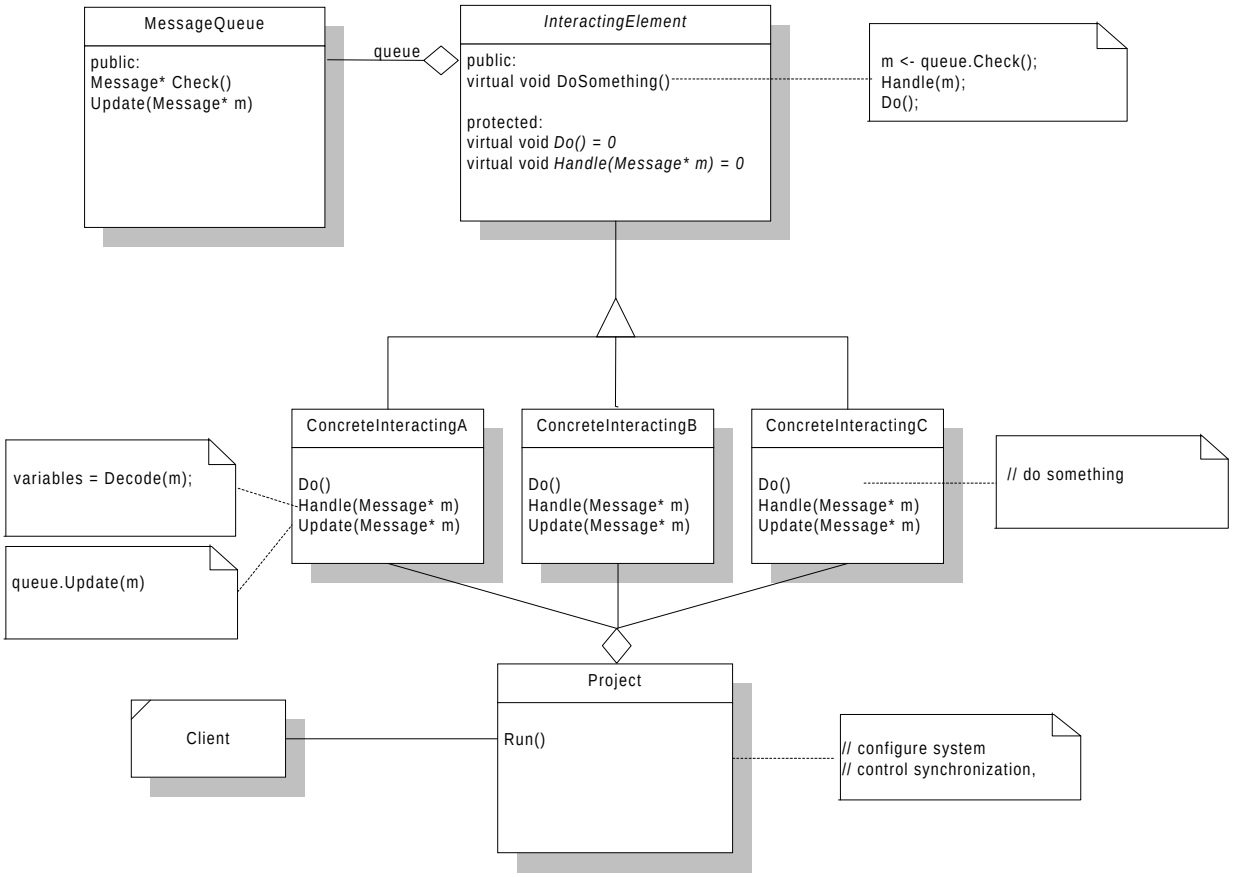

Figure 4. MULTI-AUTOMATA design structure

2. Formal semantics

The pattern formal semantics is based on the idea that every object (interacting element) has its own state diagram. Each object belongs to an initial state in the moment that it is created. In each state there are guards to control the transition firing. Each guard is a boolean expression over the object variables. The object evaluates all the state guards and, when a guard is valid, the corresponding transition is fired. For the same state, if more than one guard is valid at the time there is non-determinism, which should be solved by the pattern implementation. One possible solution is to evaluate the guards in a specific order, firing the transition of the first valid one.

Each object has to kinds of variables: local and global (that are the C++ static variables). A local variable can only be accessed by its owner object. Any active objects of a class, on the other hand, can access global variables.

Each object also has a message queue, to handle asynchronous messages. It can be specified in the language that, when an object arrives in some state it sends an asynchronous message to a specific object. In the motivation example, when the user object is created in its initial state, it sends the message "set" to the aBook object. The message queue is read only when an object is in a state in which all the guards are false. This means that asynchronous messages are handled only when the object stops in some state. Since the messages can change the value of the variables, a guard that was false can become valid and the object starts moving again. Figure 5 illustrates the elements of a class.

An interacting element class can be formally defined as:

CLASS = <Global, Local, StateDiagram, Initial, Message*>

Where:

- Global and Local represent respectively the global and local variables;
- StateDiagram is the associated state diagram;
- Initial is the initial state;
- Message* is the list of messages that can be handle by the class. Each message is defined as m ($v_1 \leftarrow c_1, \ldots, v_k \leftarrow c_k$), where m is the name of the message, the v's $\in$ (Global $\cup$ Local), and the c's are constants[2]. This means that the asynchronous messages can only change the value of the object variables.

Some constrains can be defined:

- Global $\cap$ Local = $\varnothing$;
- Initial $\in$ State(StateDiagram), where State returns all the possible states of a diagram.

Let C = <G, L, SD, I, M> be a class. Then the finite transition system that models C, can be defined as

[[C]] = <W, w_o V_i, T> where:

- W is the set of possible worlds. Each world has a state variable (st), a set of global variables (G), a set of local variables (L), and message queue (q);
- w_o is the initial world;
- V_i is a value function such that:
  - $i \in (\{st\} \cup G \cup L \cup \{q\})$;
  - V_{st}: $W \rightarrow$ State(SD);

[2] Constants are types. For the lack of simplification, the constants (syntactic elements) will be used to represent their denotation in the initial algebra of the corresponding types. The initial algebras are sub-algebras of the VALUE generic algebra.

- $V_{st}(w_o) = I$;
- $\forall v \in (G \cup L) \bullet V_v: W \rightarrow VALUE$;
- $\forall v \in (G \cup L) \bullet V_v(w_o) = v.create$, where the create method initializes the variable, assigning an initial value to it.
- $V_q: W \rightarrow Message^*$;
- $V_q(w_o) = \{\}$.
- T is the set of transitions. For each transition $t \in T$, $\alpha(t)$ is gives the source of the transition, $\beta(t)$ gives the target of the transition, and $\lambda(t)$ gives the transition's name.

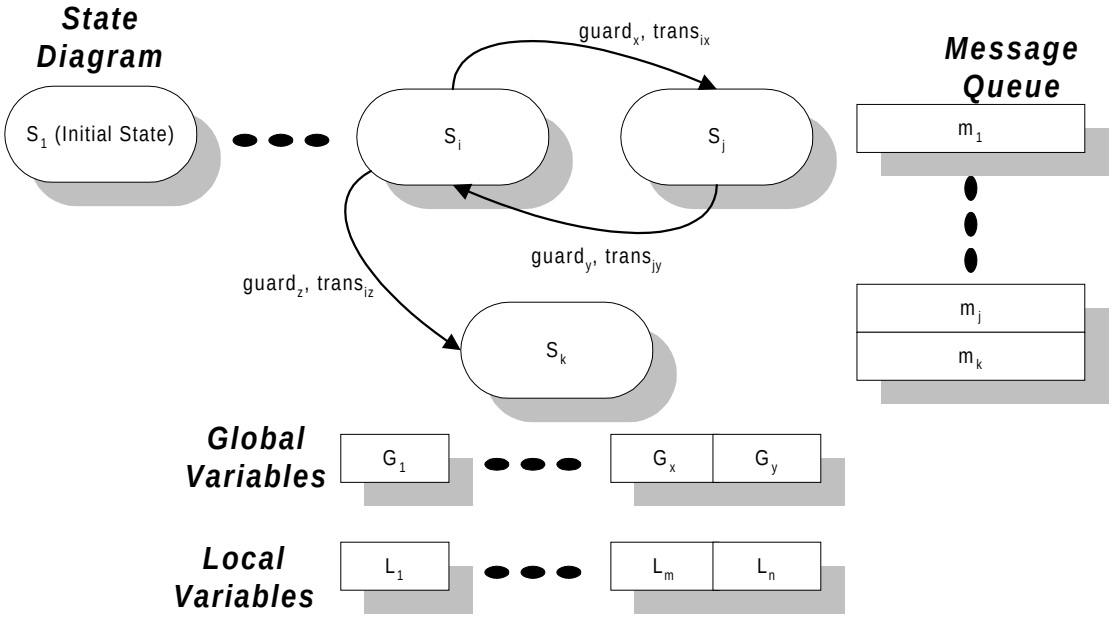

Figure 5. Modeling a class

Note that from the fact that V_{st} is a function, each object belongs to some state at any time:

$\forall w \bullet \exists s \in State(SD) \mid V_{st}(w) = s$

Figure 6 illustrates the initial world (w_o) of an arbitrary interacting element object.

Each guard can be formalized as a comparison between variables, or between variables and constants, as described bellow.

$$\forall x, v \in (G \cup L) \bullet [[v = x]]_w = \begin{cases} \text{TRUE if } V_v(w) = V_x(w) \\ \text{FALSE otherwise} \end{cases}$$

$$\forall x \in (G \cup L) \bullet [[x = c]]_w = \begin{cases} \text{TRUE if } V_x(w) = c \\ \text{FALSE otherwise} \end{cases}$$

If ϕ_1 and ϕ_2 are valid variable comparisons, we also have that:

- $[[\phi_1 \wedge \phi_2]]_W = [[\phi_1]]_W \wedge [[\phi_2]]_W$
- $[[\phi_1 \vee \phi_2]]_W = [[\phi_1]]_W \vee [[\phi_2]]_W$
- $[[\neg \phi_1]]_W = \neg [[\phi_1]]_W$

If in some world, s_i, there is at least a valid guard (comp) the object can either perform the enabled transition (trans) or receive a new asynchronous message, as illustrated in Figures 7 and 8, respectively. Note that the

message cannot be handled but it is added to the object's message queue. This behavior is formalized below.

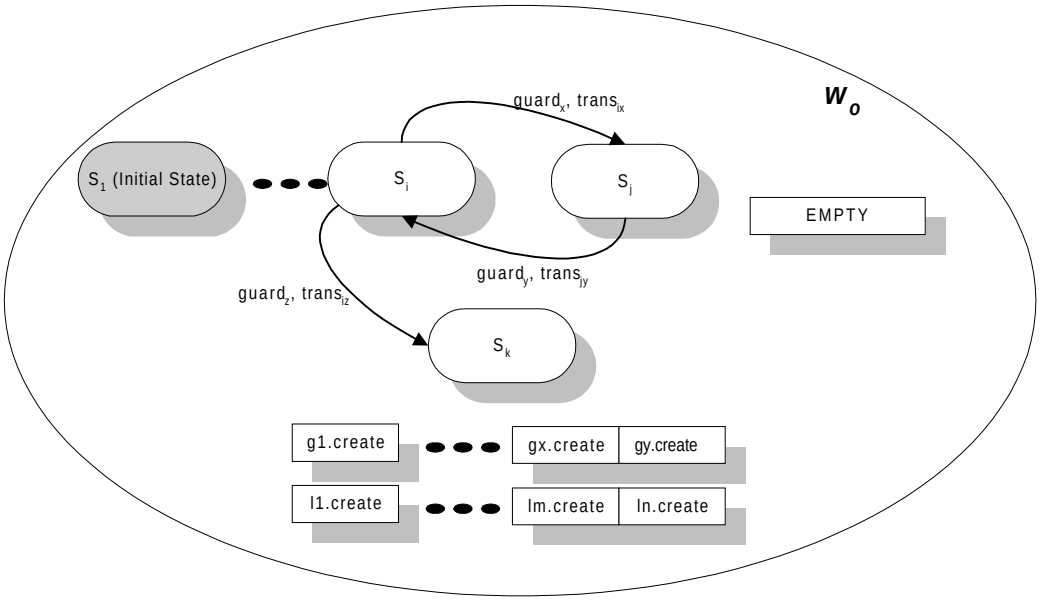

Figure 6. Initial world

$\forall$ w | $V_{st}(w) = s_i \wedge [[comp]]_W$ = TRUE •

$\exists$ t $\in$ T |

($\alpha(t) = w \wedge$

$\beta(t) = w' \wedge$

$\lambda(t)$ = trans $\wedge$

$V_{st}(w') = s_j \wedge$

($\forall$ v $\in$ (G $\cup$ L) • $V_v(w') = V_v(w)) \wedge$

$V_q(w') = V_q(w))$

$\vee$

($\alpha(t) = w \wedge$

$\beta(t) = w' \wedge$

$\lambda(t) = m(v_1 \leftarrow c_1, \ldots, v_k \leftarrow c_k) \wedge$

$V_{st}(w') = s_i \wedge$

($\forall$ v $\in$ (G $\cup$ L) • $V_v(w') = V_v(w)) \wedge$

$V_q(w') = concat(V_q(w), m(v_1 \leftarrow c_1, \ldots, v_k \leftarrow c_k)))$

On the other hand, if the object is in a world in which all the guards are false it must handle the first message of the message queue, as illustrated in Figure 9. This behavior is formalized bellow.

Suppose s_i is the source of transitions $\{(guard_{i1}, trans_{i1}), \ldots, (guard_{ik}, trans_{ik})\}$ in SD.

$\forall$ w | $V_{st}(w) = s_i \wedge (\forall j = 1..k \bullet [[guard_{ij}]]_W$ = FALSE)•

$\exists$ t $\in$ T |

$\alpha(t) = w \wedge$

$\beta(t) = w' \wedge$

$\lambda(t)$ = handle_message $\wedge$

$V_{st}(w') = V_{st}(w) = s_i \wedge$

(Head($V_q(w)$) = $m(v_1 \leftarrow c_1, \ldots, v_k \leftarrow c_k) \Rightarrow \forall$ i = 1..k • $Vv_i(w') = c_i) \wedge$

$V_q(w') = Tail(V_q(w))$

The semantics of a project object can be taken as a mapping from configurations to transition systems. A configuration is a formal device that indicates how many objects of each interacting object class are aggregated in the project class.

Let $C_1,\ldots,C_n$ be classes. We define synchronization as:

- $Sync_{ij}(C_1,\ldots,C_n) = \{<t,t'> \mid t$ is a transition in the state diagram of C_i and t' is a transition in the state diagram of $C_j\}$.
- $Sync(C_1,\ldots,C_n) = \bigcup_{k=1..n} \bigcup_{j=1..n} Sync_{ij}(C_1,\ldots,C_n)$.

Let $C_1,\ldots,C_n$ be interacting object classes, $k_1,\ldots k_n$ be positive natural numbers, and $S \subset Sync(C_1,\ldots,C_n)$. Then a project that aggregates classes $C_1,\ldots,C_n$ and has S as set of synchronization, is an structure of the form $<k_1.C_1,\ldots.,k_n.C_n, S>$. Intuitively it represents the aggregation of k_1 objects of class C_1, k_2 objects of class C_2, and so on, until k_n objects of class C_n, restricted to the synchronization set S. Each element $k_i.C_i$ is called a project component.

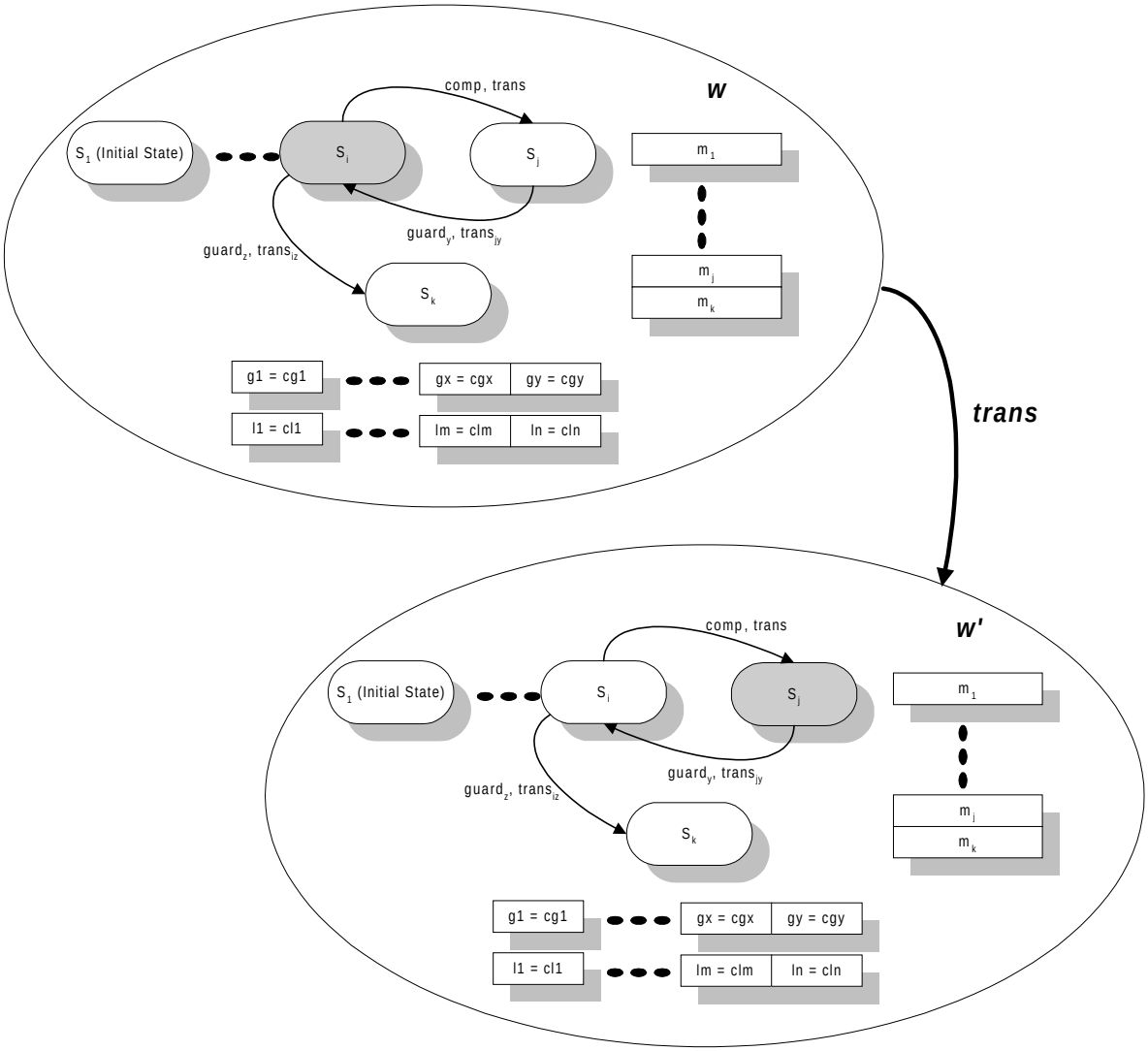

Figure 7. Change state

Let C = <G, L, SD, I, M> be a class. For any positive number j, the notation j.C denotes <G, j.L, j.SD, j.I, j.M>, where j.L= {j.v | v is in L}, j.SD denotes the state diagram obtained from SD by replacing every name state s by j.s, and, every transition pair (guard, trans), by (j.guard, j.trans), where j.guard is the comparison resulted from the replacing of variables from L to j.L. Similar renaming denotes the relationship between j.M and M, and, j.I and I.

Consider that & represents the asynchronous product of transition systems and ⊗_S, for a set of pairs of labels of transitions, represents the synchronous product of transitions system regarding the set S of synchronization pairs. The asynchronous product is defined in [1], and the synchronous product is performed by consistently re-labeling the operands in order to identify each pair of labels and performing the synchronous product as defined in [1].

The denotation of a project class is then defined as follows:

$$[[<k_1.C_1,\ldots,k_n.C_n, S>]] = \underset{i=1..n}{\otimes}\_S(\underset{j=1..k_i}{\&} [[j.C_i]])$$

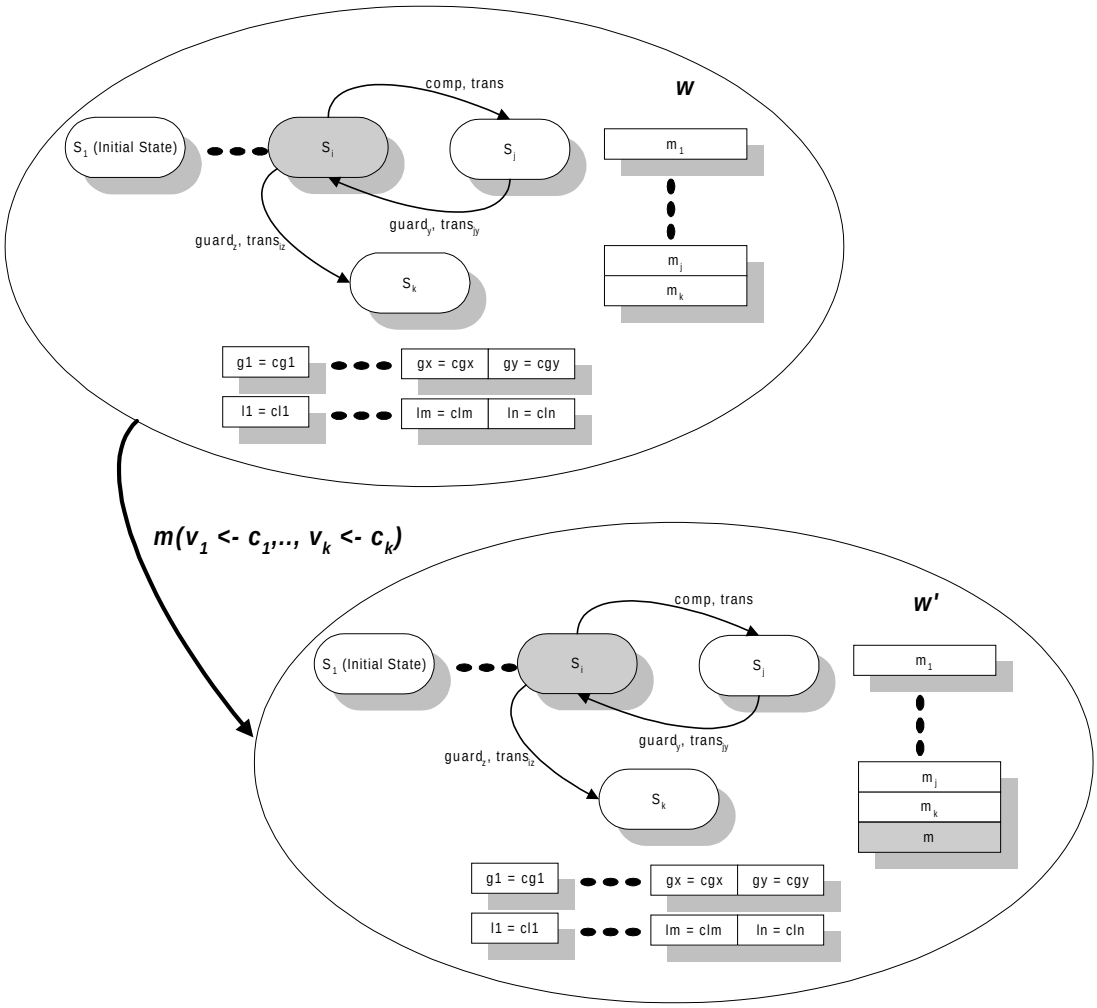

Figure 8. Receive message

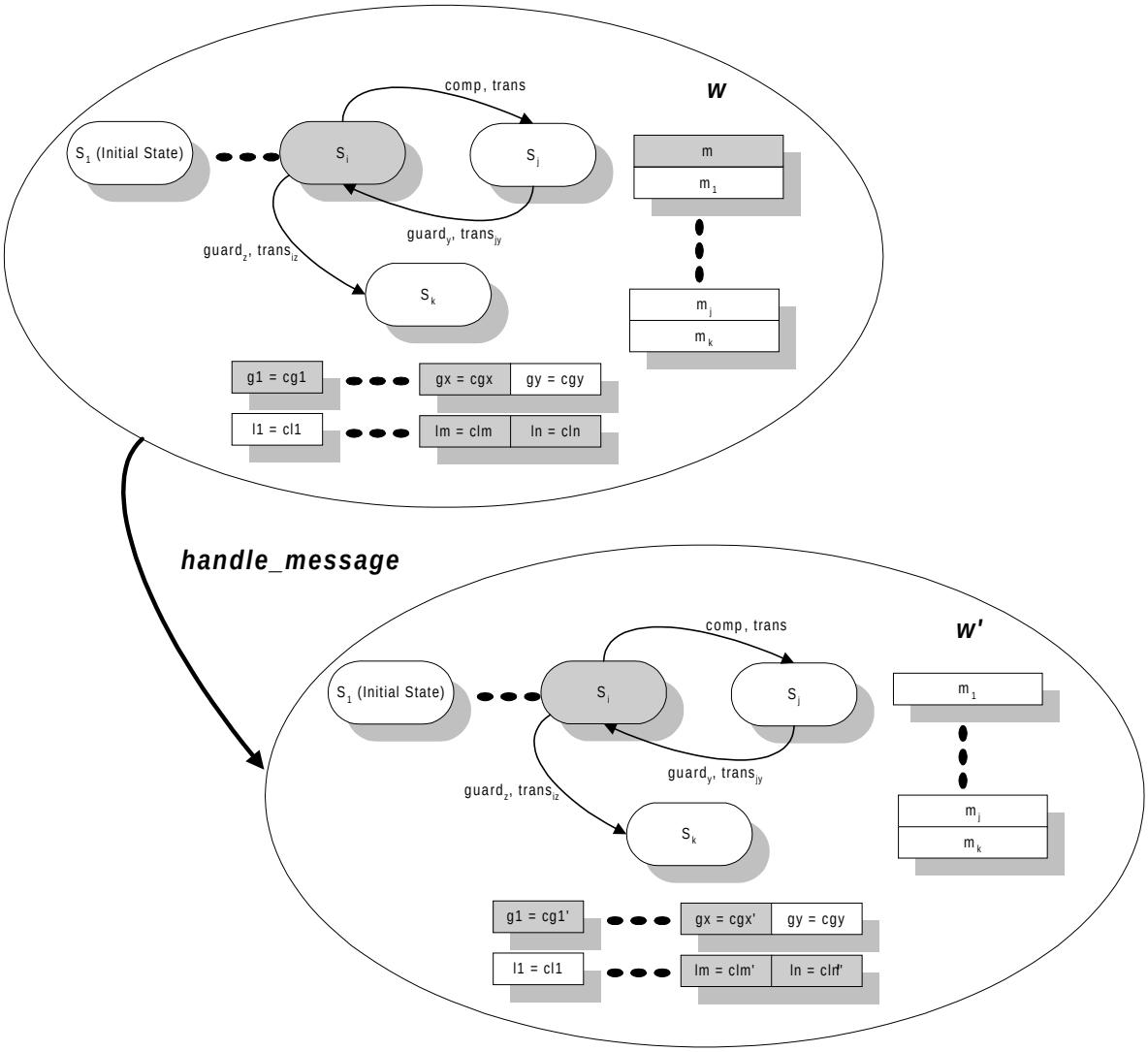

Figure 9. Handle message

3. Conclusions

The formalization presented in this paper was used as a guide to the implementation of a mapping from the MULTI-AUTOMATA pattern to SMV specifications. SMV (Symbolic Model Verifier) is a BDD model-checker based on CTL (Computational Tree Logic) and Kripke models, which can be also viewed as transition systems [4]. This mapping allows the user to generate a SMV specification from a system's configuration and find out whether certain dynamic properties, expressed by CTL formulae, hold or not. In the case the property does not hold the SMV produces a trace which shows why the property dos not hold. In [7] the whole architecture of a system that implements the MULTI-AUTOMATA pattern, namely ARTS-III, is presented.

The present paper aimed to show the pattern and its formalization. The fact that it is well suitable for Soft Real-time systems relies on the fact its semantics is based on a naturally concurrent temporal concept, namely, the very concept of Transition System. Synchrony, for example, is specified as a basic mechanism, in the same

way as synchronous product is basic in Theory of Transition Systems.

In order to extend the pattern to Hard Real-time it is enough the introduction of time limits for each transition. In this way, a Timed State Diagram expresses the behavior of an object and the semantics is based on timed transition systems as showed in [3]. Similar operation of synchronous product is defined for timed transition systems and the semantics of any project in this extension of the pattern is provided in a quite similar way. The Hard Real-time version of the pattern and its formal semantics can be used for the construction of a tool for formally based Real-time systems development. A model checker able to verify (Hard) Real-time properties (anyone based on Real-time Logic "RTL", for example) is, obviously, one of the components of this tool.

As a last word, it is worthwhile mentioning that this pattern does not concern any aspects about reliability and fault-tolerance. It provides a good design that can be formally verified by a model checker. Note that the properties to be verified typically depend on the designer of the application.

Acknowledgments

The authors would like to thank A. Haeberer, T. Maibaum, and J. Fiadeiro for their contribution to this work. Special thanks also to the ARTS-III project development team, mainly to Luiz Carlos Guedes, Juan Echaque, and Alex V. Garcia.

References

1. André Arnold, Finite Transition Systems: Semantics of Communicating Systems, Prentice Hall, 1994.
2. C. Braga, M. F. Fontoura, C. J. Lucena, and L. Moura, "Using Domain Specific Languages to Instantiate OO Frameworks", Monografias em Ciência da Computação, MCC28/98, Departamento de Informática, PUC-Rio, 1998 (also submitted to IEE Proceedings – Software Engineering, can be downloaded from http://www.les.inf.puc-rio.br/~mafe).
3. S. Carvalho, J. Fiadeiro, and E. Haeusler, "A Formal Approach to Real-time Object-oriented Software", 22nd IFAC/IFIP Workshop on Real-time Programming, Lyon, France, 1997.
4. E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen, "Symbolic Model Checking" CAV'96, LNCS 1102, 1996.
5. J. Cordy and I. Carmichael, "The TXL Programming Language Syntax and Informal Semantics", Technical Report, Queen's University at Kinkston, Canada, 1993.
6. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
7. E. Haeusler, A. Haeberer, T. Maibaum, and J. Fiadeiro, "ARTS: A Formally Supported Environment for Object-Oriented Software Development", Automating the Object-Oriented Software Development Process Workshop, ECOOP'98, 1998.
8. R. Johnson, "Frameworks = (Components + Patterns)", Communications of the ACM, 40:10, 1997.
9. J. Vlissides, Pattern Hatching: Design Patterns Applied, Software Patterns Series, Addison-Wesley, 1998.