Super Actors for Real Time

G. Fortino, L. Nigro*, F. Pupo, D. Spezzano
Laboratorio di Ingegneria del Software
Dipartimento di Elettronica, Informatica e Sistemistica
Università della Calabria, I-87036 Rende (CS), Italy

Abstract

This paper proposes a novel approach –SART- to the development of real-time systems which is based on super actors, i.e., actors whose behaviour is modelled by statecharts. SART borrows structural concepts from known methods for reactive systems like ROOM and UML-RT, and favours ease of construction by making actors reusable and composable software components.

The distinguishing factors of SART are its modularisation of timing constraints and customisable scheduling algorithm. Application actors are not aware of timing requirements. RTsynchronizers capture timing constraints in groups of actors, filter relevant messages and control their scheduling. Time clauses of causally connected messages in system interactions are directly derived from the declarative specification of end-to-end timing constraints. SART supports both prototyping and real-time execution modes.

A SART graphical development environment supporting modelling, temporal property checking and code generation has been implemented in Java.

1. Introduction

The aim of this work is to provide a methodology – SART- and a graphical environment for modelling, prototyping and implementation of distributed real-time (RT) systems. SART is centred on the concept of *super actors* whose behaviour is specified by statecharts [1]. Although several and powerful modelling languages and visual tools based on statecharts have been proposed for RT systems, e.g., ROOM [2], STATEMATE [3], UML-RT [4], this work claims that the explicit support of *timing constraints*, which are essential for RT design, from the viewpoints of specification, validation and enforcement, remains unsatisfactory. There is normally a semantic gap between high-level development phases and low-level final implementation phases of a project where a modelled system is ultimately managed by a real-time operating system whose (hidden) scheduling algorithm, normally through priority and pre-emption, is responsible for the fulfilment of all the timing requirements. A high-level notation often only makes it possible to constrain an activity with the passage of time through a timer concept, this is of limited effect for dealing with system reactions to external stimuli [5]. Recently [6] some constructs have been proposed for UML-RT for the purposes of specifying and analysing end-to-end timing constraints in system response.

The approach adopted in SART and described in this paper is novel in that it favours timing predictability by integrating an application with its runtime executive. The key feature of SART is modularisation of timing constraints [5]. Application actors are not aware of timing constraints. RTsynchronizers [7-9] are introduced which capture timing clauses of individual events in system reactions and regulate its message-based scheduling. Rtsynchronizers are derived from a synthesis algorithm [10-12] which operates on an RTL-like [12] specification of the timing constraints. The modelling language of SART is close to existing notations, particularly the component-based architecture of ROOM and UML-RT. Actors are developed as reusable components which can be linked to one another by interface ports with associated input/output messages to generate new components and so forth. Specific contributions of SART are: (a) a timing constraints synthesis and mechanization of Rtsynchronizers; (b) a customisable scheduling algorithm; (c) a concept for modularising groups of actors (as a subsystem or cluster) on the basis of shared timing interactions and constraints; (d) a minor simplification to the statecharts formalism which improves modeller activities; (e) a system life cycle which unites modelling, analysis, design and implementation phases; (f) a graphical development environment which enables system modelling, property checking by prototyping/simulation, and automatic code generation in Java.

This paper gives an overview to SART by describing its computational model, modelling language and management of timing constraints. Finally, the implementation status together with an indication of ongoing work are briefly discussed in the conclusions.

* Corresponding author - phone: +39-0984-494748, l.nigro@unical.it

2. Computational Model

SART represents an eclectic approach. It follows the ideas of the Modelling Dimensions Paradigm [2] and builds on concepts developed within the community of the Actors model [14,9,15,5]. The approach is three-dimensional and focuses on Structure, Behaviour and Timing Constraints which can separately be dealt with during a project.

A UML-like style is adopted for modelling the structure and behaviour of the components of a system. The behavioural part of components can be hierarchical organized according to a statecharts formalism. Actor components only provide functional issues, i.e., the actions for responding to incoming messages.

Rtsynchronizers [7-9,15,12], i.e., specialised actors, are responsible of the timing issues in group of actors and have a reflective link with scheduling. Actors communicate to one another by asynchronous message passing. An actor responds to a message by making a state transition and executing an action. Action execution is atomic. Actors are non thread objects. They are at rest until a message arrives. After responding to a message, the actor is ready to accept the next message etc.

Actors can be grouped into clusters. Each cluster is governed by a control machine which provides the basic scheduling and dispatching activities. Actor concurrency within a cluster is co-operative and not pre-emptive. Each cluster can be allocated to a distinct network node of a distributed system. Messages represent the scheduling units. They transparently have timing constraints associated with them by Rtsynchronizers thus affecting the message selection and delivery process.

The basic actor architecture is summarised in Figure 1. The resultant architecture purposely avoids any dependency on a predefined RT OS and associated concurrency control mechanisms.

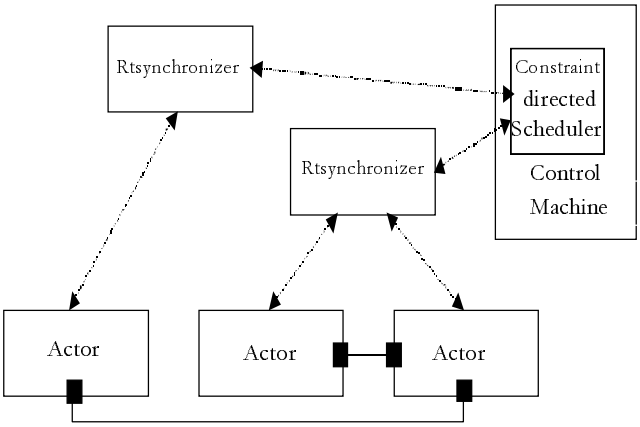


Figure 1: Basic actor architecture

3. Modelling Language

The SART modelling language was influenced by known notations of ROOM[2] and UML-RT[4], particularly for the structural and behavioural aspects. For instance, the actor component closely corresponds to the *capsule* of UML-RT. The following outlines the modelling language by focussing on the differences introduced with respect to existing notations.

3.1 Structure

A system consists of a set of interconnected actors (see Figure 1). As in ROOM and UML-RT, actors have an interface of typed ports. A port is characterised by the set of messages which can flow through it. Messages can be input or output and can carry data parameters. For simplicity, ports have no protocol rules. Two ports P_1 and P_2 with respectively input messages I_1 and I_2 and output messages O_1 and O_2 can be linked to one another if they have *compatible messages*, i.e., $O_1 \subseteq I_2$ and $O_2 \subseteq I_1$.

Component based design is supported by having actors which are turned into reusable components which can be interconnected to generate new actors and so forth. To facilitate actor aggregation the concept of relay ports and behaviour (or end) ports are introduced as in ROOM. Actor aggregation with behaviour specialization can be used to simulate actor inheritance. Reusable actors can be imported from a library and be instantiated in different projects. However, no primitive notion of array of actors is provided. Structural aspects of an actor include a data environment, possibly hierarchical structured, which can only be modified by responding to messages on the basis of the rules expressed by the actor behaviour.

3.2 Behaviour

The dynamic behaviour of an actor is modelled by statecharts [1] according to the *or*-decomposition style like in ROOM [2] (see Figure 6). From time to time an actor can find itself into one of a set of disjoint states. However, state hierarchy implies that at a given instant in time the actor resides in a state and all of its enclosing (super) states up to the top state. Concurrency is supported at the actor level not within states (*and*-decomposition). All of this complies with the adopted actor computational model and the basic goal of achieving a timing predictable framework intended to be mechanically transformed from graphical specification into design and implementation. In addition asynchronous point-to-point message passing is used instead of shared data and broadcast communications.

Several minor modifications/additions were considered to the graphical modelling of statecharts. Although they do not add expressive power to the formalism, they can be

useful for incremental development. Besides the elimination of *fork*/*join* pseudo states relevant to *and*-decompositions, also the pseudo state *end state* is removed since in the assumed actor model every state can behave as an end state. The pseudo states *start state*, *shallow history* (H) and *deep history* (H*) are retained unchanged. States can be macro states (or super states) to permit decomposition, leaf states otherwise. State diagrams follow the Mealy style with transitions which are labelled, in general, by the trigger message, the associated guard and the corresponding action which is executed when the transition fires. In addition, as in ROOM, an *entry action* and an *exit action* can possibly be added to every state. Through a convenient use of conditions it can be simplified the drawing of transitions between internal states of macro states and external states and vice versa. Figures 2 to 4 exemplify the effect of transition rewriting between macro states. The final result is that any such a transition can always be drawn *between* macro states, thus eliminating *chain states* [4] and facilitating the stepwise refinement of macro states.

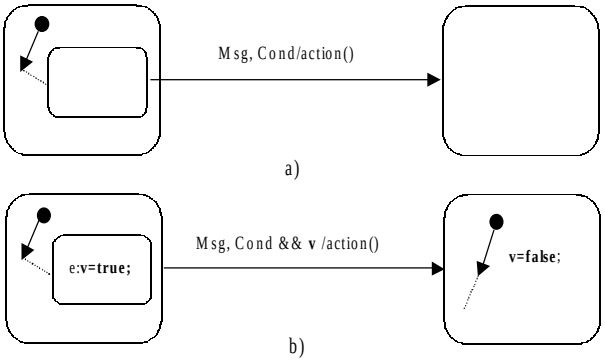


Figure 2: Transition rewriting 1, a) original, b) modified

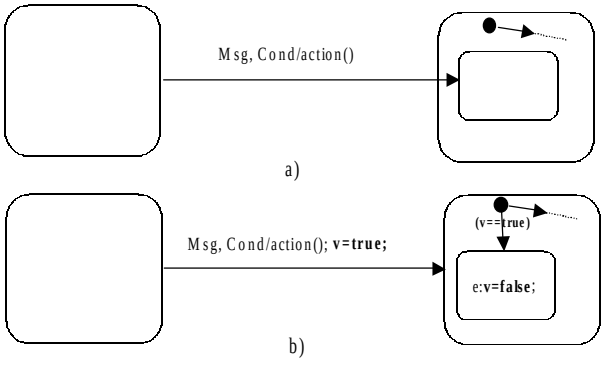


Figure 3: Transition rewriting 2, a) original b) modified

The concept of immediate transition has been introduced which makes it possible to eliminate branch-states. However its use is general. An immediate transition is one which after being executed directly consigns its triggering event to the reached state which immediately processes it (without scheduler intervention). Immediate transitions simplify the description of exceptional events or interrupts, by avoiding the generation of unnecessary messages. Branch state elimination is illustrated in Figure 5.

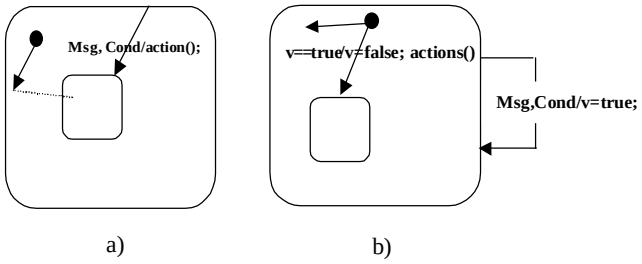


Figure 4: Transition rewriting 3, a) original b) modified

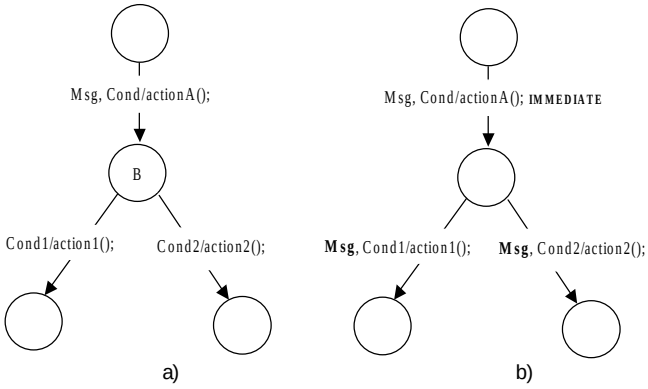


Figure 5: Branch state (B) elimination a) original b) modified

Another minimal added feature concerns the use in macro states of multiple *input transitions* (e.g., see Fig. 3b), each with an associated guard condition, exiting from a start state, with a default transition which is taken when no other guard evaluates to true. Figure 6 portrays the shape of a typical SART actor behaviour.

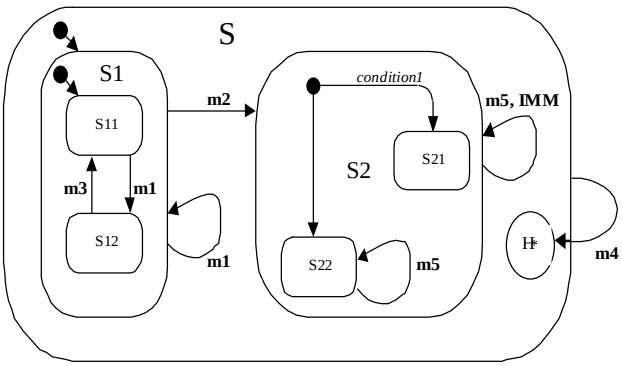


Figure 6: An actor state hierarchy

Messages m1, m4 and m5 are triggers of *group transitions*. However, whereas an occurrence of m1 or m5 forces the relevant macro state to be entered through its pseudo start state, an m4 occurrence corresponds to an *interrupt*. After some computation (action associated to m4 transition), control re-enters (deep history) the state within S which was left off at the time of m4 arrival. Message m5 triggers an immediate transition. The internal state of S2 (S21 or S22) selected by the enabled input transition is then given the immediate chance of process message m5. When a message is received in a state which is unable to provide a response to it (e.g., the arrival of m4 when it is current S11 of S1), it is propagated to its immediately enclosing super state (e.g., S1) and so forth until either a response is available (e.g., in the top state S) or the message is eventually rejected. Figure 6 exemplifies the concept of *transition overriding*. When current (sub) state is S11 of S1 and message m1 arrives, the internal transition in S1 is chosen instead of the group transition of S1.

3.3 Timing Constraints

SART maintains a separation of concerns for meeting the functional and real-time requirements of a system. Functional requirements are dealt with by drawing actor structure and behaviour. Timing constraints existing in system reactions are separately modelled through a declarative style which was inspired by the Real Time Logic (RTL) [13,10-11] formalism. The notation naturally adhers to the event-driven actor computational model and the role of actions, i.e., atomic message processing, which provide functions and represent the scheduling units. The concept of *temporal activity diagram* (TAD) was introduced for capturing the causal effect relationship among events (or messages) and the associated timing information. TADs are message-based threads of control and serve for the synthesis of timing constraints, i.e., deriving the time clauses for all the involved events in system reactions. They also help checking/fixing timing violations and support the mechanical construction of Rtsynchronizers which impose timing constraints to messages during runtime. The validation of system temporal properties is in any case deferred to a subsequent prototyping/simulation of the modelled system, where the effect of interleaving multiple system reactions can be observed under different load conditions. In the following, a discrete domain of global real time will be assumed.

A TAD is a graph model where nodes are associated to event occurrences and directed edges mirror the causal relationship among events. Each TAD is drawn for a single reaction. Both nodes and edges are annotated with temporal information (see Figure 7). An event node carries a triple <*start time, processing time, deadline*>. An edge is attached a time interval $[L_b..U_b]$, $L_b \leq U_b$, which can express, e.g., in a distributed framework, a delay (or transfer time) in the occurrence of a caused event since the occurrence of the corresponding source event. In Figure 7 node A has starting time S_A, computation time P_A and deadline D_A whereas node B has the corresponding parameters S_B, P_B and D_B. By convention, an edge without any annotation has time interval [0..0] (instantaneous causal event connection); a single time value d is equivalent to [d..d]. Figure 7 shows the basic *precedence* construct in TAD's and corresponds to the $<$, $\leq$ RTL operators. It states that action B is caused by action A and that, in RTL terms, $(\uparrow B \geq \downarrow A + L_b) \wedge (\uparrow B \leq \downarrow A + U_b)$, where symbol $\uparrow$ indicates event starting and $\downarrow$ event ending. In other words, $\downarrow A + L_b \leq S_B \leq \downarrow A + U_b$. Other primitive constructs are summarised in Figure 8. However, to facilitate the construction of TAD's some non primitive constructs can also be used which are short cuts of basic constructs interconnection. The most common abbreviations are portrayed in Figure 9. They are respectively associated to a group of events which are spawned by a same source (RTL $\wedge/\vee$ operators) and the case of a single event which acts as the synchronization point of a group of causal events. The first one can have the AND logic or the OR/XOR interpretation. In Figure 9b) only the AND equivalence is shown, it is based on the equality node followed by the spawned concurrent actions. However, the use of conditions on the spawned actions can make the outgoing branches OR paths like in Figure 8d) or XOR paths where only one branch is actually followed depending on the value of the conditions (see Figure 10 for an example).

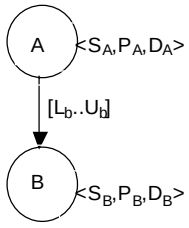
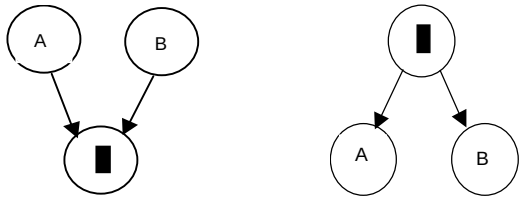
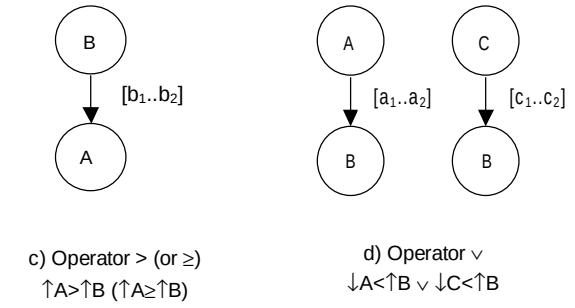


Figure 7: Basic TAD precedence construct



a) Equality node, $\downarrow A \wedge \downarrow B$ b) Equality node, $\uparrow A == \uparrow B$

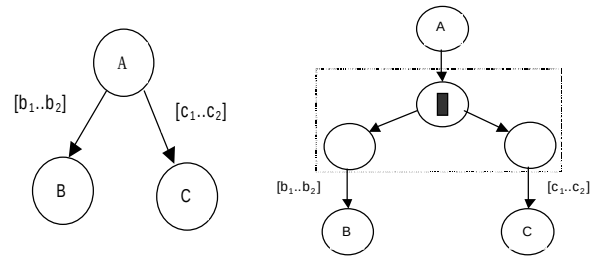
Figure 8: Other primitive TAD constructs



c) Operator > (or ≥)
↑A>↑B (↑A≥↑B)

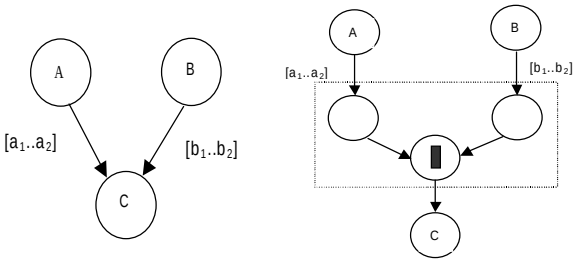
d) Operator ∨
↓A<↑B ∨ ↓C<↑B

Figure 8: Other primitive TAD constructs (continued)



a) Spawning

b) Spawning AND equivalence



c) Synchronization

d) Synchronization equivalence

Figure 9: Non primitive TAD constructs

It is worth noting that the existence of TAD constructs for both <, ≤, and >, ≥ RTL operators conveniently avoids considering negative relative times in timing inequalities during declaration and synthesis of timing constraints.

A group of events not related by precedence constraints but only on the global time, e.g., A+B+C<T, can be re-written simply as ↓A<T∨↓B<T∨↓C<T without considering the 3! possible cases implied by the original expression. All of this is a consequence of the underlying scheduling model which guarantees events are selected according to their timing windows, supposed these timing windows are admissible.

The timing constraints synthesis algorithm operates on the collection of TAD's which describes the temporal behaviour of a system. TAD's are initially annotated by start time and deadline in input nodes, which trigger system reactions, and the processing time in every other event node. Processing times can possibly be omitted in early analysis but must be included for final calculation of time clauses. Start time and deadline of remaining nodes are respectively assigned the default value of 0 and ∞. The algorithm basically propagates timing windows from input nodes down to causally dependent nodes (*forward propagation*).

Referring to the primitive precedence construct in Figure 7 the propagation rules are the following:

$$S_B=\max(S_A+P_A+L_b, S_B)$$
$$D_B=\min(D_A+U_b, D_B)$$

However, the presence of equality nodes as in Figure 8a) can require a *backward propagation*. If $<S_{EQ},0,D_{EQ}>$ is the time window associated to the equality node, and D_A and D_B are the deadlines of incident nodes A and B, the backward propagation rule simply assigns to nodes A and B the deadline of the equality node: $D_A=D_{EQ}$, $D_B=D_{EQ}$.

The following exemplifies an application of TADs to a typical control system consisting of four actors: a Sensor, an Actuator, a Display and a Controller. The Sensor samples data from the external environment every 50 time units (TU) and transmits it to the Controller which must react through the Actuator with either a normal or an emergency response but not both within 45 TU after receiving the data.

Two TAD's respectively associated to normal response and emergency response are portrayed in Figure 10. They share the Data Sampling (DS) node. The Controller selects the response's TAD by a data check (*Cond*) carried in the Transmit Data (TD) action. In the normal TAD the data are displayed on a user screen whose background is prepared (Display Preparation, DP) while the Controller makes Data Analysis (DA). After both DP and DA, the analyzed data are actually displayed (Data Display, DD) and finally the Normal Response (NR) is generated. In the emergency TAD data are only analyzed (DA) and then the Emergency Response (ER) is provided. Normal and emergency TAD's are annotated by the assumed processing times and the required deadline (see the edges DS-TD). Time information in Figure 10 is relative to the initial instant of the sampling data.

It is worthy of note that each TAD implicitly follows the universal ∀ operator of RTL. Indeed, each time the DS event occurs, the temporal activity is re-started from its beginning and the relative time is purposely reset as usually is useful when modeling periodic system reactions. The Sensor actor is assumed to have a functional behavior where the DS message is self-sent after being received.

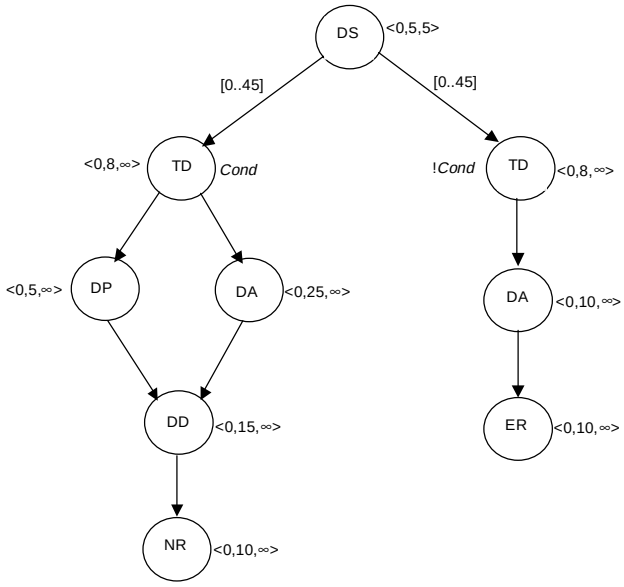


Figure 10: TAD's for the example control system

Applying the synthesis algorithm to Figure 10 will update the timing constraints as shown in the Figure 11. The time window of DD node is the result of left and right propagation paths. Although the left propagation path would associate the window <18,15,50>, the right path, according to the propagation rules, replaces it with the final window <38,15,50>.

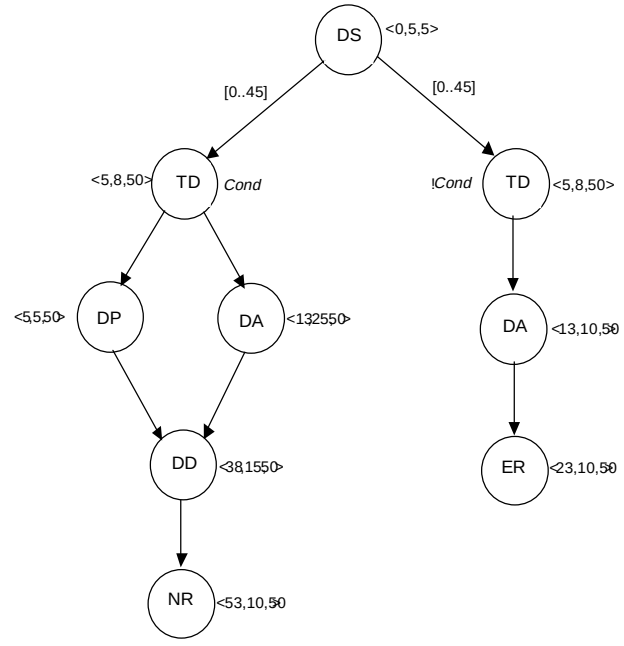


Figure 11: Effect of the application of the synthesis algorithm

Given the initial time parameters, Figure 11 clearly reports a failure in fulfilling the deadline during a normal response, since NR starts at 53 and doesn't end until 63. Therefore, either the sampling period should be augmented or the processing time of critical actions (e.g., DA) must be shortened or the both.

3.3.1 Mechanization of Rtsynchronizers

An Rtsynchronizer is a special purpose actor which transparently interfaces a group of controlled application actors with the runtime executive (scheduler). It gets involved and carries actions at both the send time and the dispatch (receive) time of a message. To exemplify, at the send time of a message M the usual action executed by the Rtsynchronizer is the following:

```
if( message is M && condition )
    schedule( M with M'time window );
```

where M is assumed to be sent by a sender actor which associates to it a given condition (e.g., stating the current internal state of the sender). The time window of M is defined by the synthesis of the timing constraints described in section 3.2. At dispatch time, the Rtsynchronizer can memorize the occurrence time of the message and so forth.

The behaviour of a Rtsynchronizer is a flat state model and consists of a single state with self transitions labelled by relevant Condition/Action pairs. As a consequence, the design of Rtsynchronizers can be mechanized. The actual Temporal Activity Diagram (TAD) modelling language of SART, as supported by the graphical environment, helps automating the derivation of Rtsynchronizers by using the modelling objects portrayed in Figure 12.

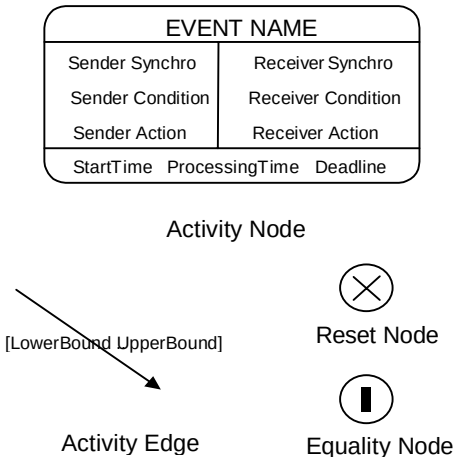


Figure 12: Temporal Activity Diagram graphical notation

The Activity Node describes an event/action occurrence node in a TAD and specifies, besides the event name and the associated time window, the names of the sender/receiver Rtsynchronizers. As a particular case the two can be a same Rtsynchronizer. The relevant pairs Condition/Action are reported for the two synchronizers. The expression of the Condition and the Action relies on a high-level programming language (e.g., Java). The Activity Edge is accompanied by the time interval required by the timing constraints propagation algorithm. The Reset Node can be useful in complex TAD's for explicit reset of the relative time marking the starting of a sub-activity.

3.4 Scheduling

A key factor of SART is its ability to work with a custom scheduling algorithm. Since the absence of pre-emption, finding in general the optimal schedule for the event set of a SART system is a well-known NP-hard problem. Therefore, concrete schedulers can be achieved by using some heuristics at runtime, i.e., at each message send. In order to characterize the implementation of one such a scheduler, the following terms are useful (see Figure 13). At a given instant in time *n* messages (events) are pending within the scheduler. Let t_{si} be the start time, t_i the effective starting time, d_i the deadline and p_i the processing time of a message m_i.

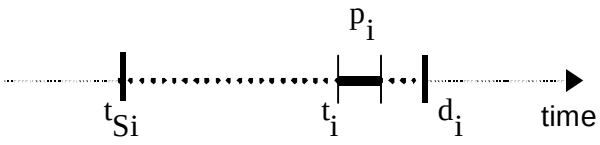


Figure 13: Message timing constraint representation

The chosen heuristic is based on the following precedence relation "$\prec$" between messages:

$$m_i \prec m_j \Leftrightarrow d_i\text{-}p_i < t_{sj}+p_j$$

Indeed (see Figure 14), would it be true the timing inequality between m_i and m_j, choosing m_j would forbid m_i from meeting its timing constraint. The "$\prec$" precedence relation makes it possible to reduce the number of precedence constraints in the formal definition of the makespan optimisation problem. The scheduler dynamically selects the "most critical" pending message for dispatching, i.e., that one which has minimum d_i-p_i.

Operatively, the implemented scheduler uses two lists: L_D and L_S. L_D stores messages ranked by ascending d_i-p_i, i.e., the maximum admissible delay for m_i. L_S is kept ranked by ascending $t_{sj}+p_j$, i.e., the minimum end time for message m_j. The algorithm first builds the candidate set S_C made up of messages in L_S which have rank less than L_D'first. In the case S_C has one element, this one constitutes the dispatch message. If S_C is empty, the dispatch message coincides with L_D'first (a decision EDF like). If S_C has more than one element, the message is selected (this is just one possibility) which has minimum t_S. This choice reduces the idle times of the CPU and contributes to shortening the overall cost function of the optimisation problem.

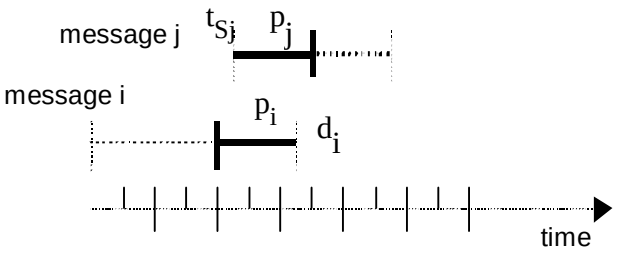


Figure 14: Precedence relation between messages

The scheduler algorithm can also be adapted to operate in simulation mode, with time which is explicitly advanced at message dispatching and with worst-case processing times which are assumed for messages. The prototyping/simulation execution mode is useful for checking timing constraints in the presence of multiple and conflicting system reactions. The application remains basically the same whereas several scenarios (load conditions) can be considered for driving the temporal testing phase.

4. Conclusions

The development of real-time systems can greatly benefit from the software engineering advantages of powerful modelling languages based on component-based design and statecharts. However, to be effective, a notation should allow the explicit specification of and reasoning on the timing constraints which exist in system reactions and thus determine the design. SART is an eclectic approach presented in this paper which borrows structure and behaviour concepts from state of the art modelling languages like ROOM [2], STATEMATE [3] and UML-RT [4] and integrates them with a timing framework which depends on an RTL [13] graphical style of timing constraints specification, Rtsynchronizers [7-9] and custom scheduling are used for the analysis and enforcement of the timing requirements.

SART has a synthesis tool which is able to propagate/analyse timing information within the message-based thread of control which constitutes a system

reaction and thus automate the generation of Rtsynchronizers.

A toolset in Java2 has been designed which delivers a full system life cycle based on SART. Functional and temporal modelling, property validation through simulation and implementation with automatic generation of Java code are supported. The various phases of a system development are united in terms of the common underlying actor representation. This in turn minimises the risks of introducing errors when transforming a specification into a final implementation. Separate but integrated tools are provided for:

(a) designing the structure of actors and their port interconnections to form new components or systems;
(b) modelling actor behaviours through proposed statecharts;
(c) specifying timing constraints in system activities and mechanizing the production of Rtsynchronizers;
(d) linking Rtsynchronizers to actors also in the context of a distributed implementation of the system.

Each tool can generate independent code. An implemented system can use a reduced Java Virtual Machine where, for instance, garbage collection is avoided by having messages which are collected into reusable pools from where they are picked-up on demand without necessarily using the new operator. In addition, only one Java thread is used to support the scheduler operation and the atomic execution of actions in actors.

On-going work is directed at improving the toolset and completing it with a support for distribution aspects and to experiment with the use of the developed tools in complex real-time system design.

Acknowledgments

The authors wish to thank Brian Kirk for the helpful discussions during the preparation of the paper.

References

[1] Harel D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming* 8, pp. 231-274.

[2] Selic B., G. Gulleckson and P.T. Ward (1994). *Real-time object-oriented modelling*. Wiley.

[3] Harel D. and M. Politi (1998). *Modeling reactive systems with statecharts*. McGraw-Hill.

[4] Selic B. and J. Rumbaugh (1998). Using UML for modelling complex real-time systems. http://www.ObjecTime.com/otl/technical, O*bjecTime Limited*.

[5] Kirk B., L. Nigro and F. Pupo (1997). Using real time constraints for modularisation. *Lecture Notes in Computer Science*, 1204, Springer-Verlag, pp. 236-251.

[6] Hermeling M., O. van Roosmalen and B. Selic (1999). Timing constraints and object-oriented design. *Proc. of 24th IFAC/IFIP Workshop on Real Time Programming (WRTP'99)*, pp. 15-20.

[7] Ren, S., Agha, G. (1995): Rtsynchronizer: language support for real-time specification in distributed systems. *ACM SIGPLAN Notices*, **30**, pp. 50-59.

[8] Saito, M., Agha, G. (1995): A modular approach to real-time synchronisation. In *Object-Oriented Real-Time Systems Workshop, OOPS Messenger, ACM SIGPLAN*, 1995.

[9] Agha, G. (1996): Abstracting interaction patterns: a programming paradigm for open distributed systems. *Formal Methods for Open Object-based Distributed Systems*, Vol. 1, (Najm E. and Stefani J. B., Eds.), Chapman & Hall.

[10] Mok A. K. (1991). Towards mechanization of real-time system design. In *Foundation of real-time computing: formal specification and methods*, (A.M.V. Tilborg and G.M. Koob, Eds.), Kluwer Academic Publishers, pp. 1-38.

[11] Tsai J.J.P., S.J. Yang, Y.-H. Chang (1995). Timing constraint Petri nets and their application to the schedulability analysis of real-time specifications. *IEEE Trans. Soft. Eng.*, **21**(1), pp. 32-49.

[12] Nigro L. and F. Pupo (2000). Schedulability analysis of real time actor systems using Coloured Petri Nets. To appear in *Advances in Petri Nets* series of Lecture Notes in Computer Science, special volume *Concurrent Object-Oriented Programming and Petri Nets* (G. Agha, F. De Cindio and G. Rozenberg Eds.)

[13] Jahanian F. and A. K. Mok (1986). Safety analysis of timing properties in real-time systems. *IEEE Trans. Soft. Eng.*, **12**(9), pp. 890-904.

[14] Agha G. (1986). *Actors: A model for concurrent computation in distributed systems*. MIT Press.

[15] Nielsen B., S. Ren and G. Agha (1998). Specification of real-time interaction constraints. *Proc. of First Int. Symposium on Object-Oriented Real-Time Computing*, IEEE Computer Society.