Computer Science Journal of Moldova, vol.**19**, no.3(57), 2011

Concurrency specification using Event-based Specification Chart

Dumitru Ciorbă, Victor Beşliu

Abstract

Architecting framework proposed in [1] can be used efficiently for developing concurrency-intensive systems only if there exist languages and tools corresponding to the described concepts. In this article there will be presented an approach based on using formalism. Theoretical advantages of formal specification are well known. However, usage of formal specification in practice ascertains some difficulties, thus their current advantages are not widely explored. The main focus of our research is to improve usage of formal method in verification of concurrency. Our vision consists in adapting the pragmatic approach and relaxing formalism, by creating graphical specification language based on events.

Keywords: software architecture, concurrency, formalization, specification, CSP#

1 Specification and formal methods

It is well known that methods of formal specification allow [2, 3]: to describe completely behavior of a system; to analyze in detail and, consequently, to better understand systems; to facilitate verification, maintenance and development of system; to reduce number of errors, etc.

In practice, the methods of formal specifications are not widely used due to the fact that formal character of specification languages makes difficult their understanding and usage.

Usually, overall development time using formal methods is expanded so much that the question arises whether to use them or not. The work [3] reflects unavailability of the majority of IT-experts to the methods of formal specifications and disadvantages of full formal analysis of large and complex systems. Despite this, the author of the work [3], being a proponent of formal specification methods, insists that the key to achieving good results is exactly the use of formal methods in early stages of development. Numerous studies [3, 4, 5] have shown that the earlier discrepancy is detected, the cheaper the error costs (Figure 1).

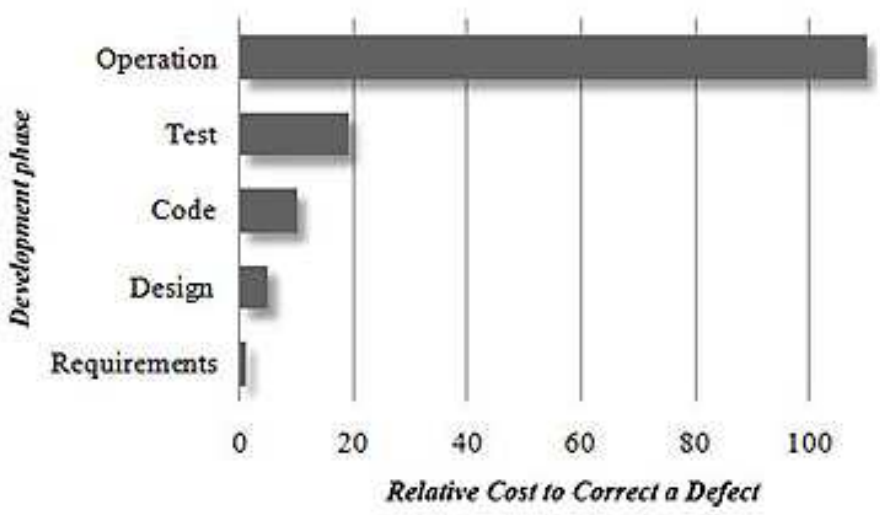

Figure 1. Relative cost to correct a defect

But is it always justified the effort for the formal specification of the system in the early stages of development? In [2] it is insisted that transition from informal requirements to formal specifications should not be done too early because of the fact that a greater degree of detail complicates the specification.

Also, in the above-mentioned work, it is proposed *a pragmatic ap-*

proach to formal specification, which consists in usage, if necessary, of several formalisms for specification of different system aspects, increasing expressiveness of formalization, in general, and avoiding restrictions of specification languages, in particular.

The same approach applies when using UML (Unified Modeling Language, [6]). However the language is semi-formal, even in the case of model annotating with OCL-expressions (Object Constraint Language, [7]). In addition to UML and ADL (Architecture Description Language) diagrams, at architectural level, for describing system behavior, the formal language needs to be used. Anyway, without doubt, today it is best used in the IT industry.

The expressiveness of UML models, the "standardization" of the software development processes, the advantages of the methods of formal specifications and pragmatic approach, incline to improve the specification phase through complement or extension of existing methods rather than through development of new universal and common language for formal specification.

2 Component-based structure architecting

Structure architecting consists in describing the set of components and set of connections (connectors) defining components interaction. Structure view of architecting process focuses on term, which is closely related to system topology.

Description of architecture as configurations of components and connectors is very popular, and, in majority of cases, architectural description languages (ADLs) graphically represent them as "boxes and lines".

The ACME language of architecture description must be mentioned. ACME is a formal meta-language, which provides a set of language constructs for describing architectural structure, architectural types and styles, and annotated properties of the architectural elements [8].

Publish-subscribe style architecture is represented in Figure 2 using ACME-like concepts and graphical representations.

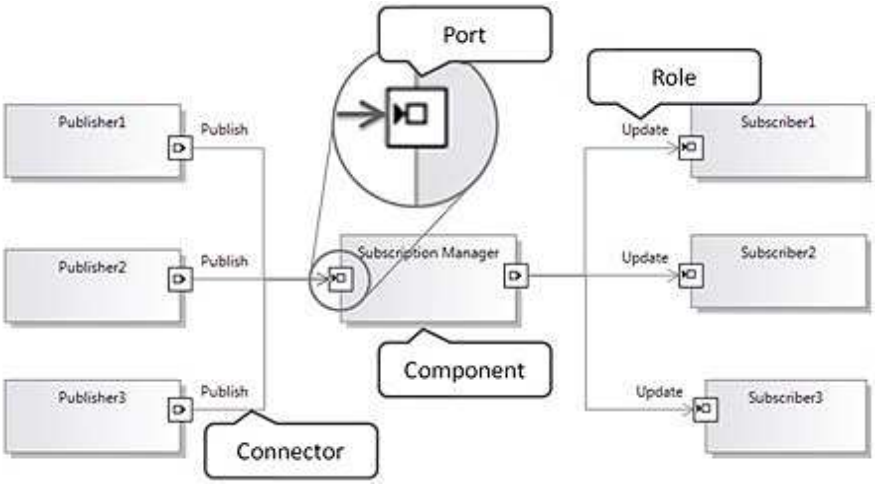

Figure 2. Structure view of a publish-subscribe architecture

Component is an elementary system entity, which represents process or data store unit of a system. Thus, components are abstractions used to model hardware and software elements. A module, library, class or other encapsulation unit can be an analogue of a component, abstract description of which can be captured by *properties*. Component functionalities are exposed through *ports*. Thus, port represents the interface which describes services (operations) of components that are provided or requested

Connector makes interaction as explicit concept. All architecture connectors embody a protocol of communication and synchronization between participants, which are defined by *roles*. Connectors have properties too. Details of interactions can be specified exactly using properties and another formalisms and tools as "value" (e.g. CSP-like language Wright or described bellow ESC language). Connectors and ports can be discoverable at run-time, and define a "transport" independent communication between components.

3 Event-based behavior architecting

3.1 Event centric development

Event-based behavior architecting permits very flexible interactions between system active entities, therefore event-driven techniques remain popular for concurrency for a long time. *Matt Welsh* [9] had adopted an event-driven architecture too, in order to support massive degrees of concurrency. Resulted *staged event-driven architecture* (SEDA) is very efficient and has a robust structure.

Event centric architecture in the simplest case defines four basic entities (Figure 3):

Figure 3. Basic entities of the event-based architecture

- Data (what data is processed);
- Actions (actions taken when processing);
- Components (where actions of processing localized);
- Events (when and in what sequence are actions activated).

Inexpensive synchronization is an important argument for event-driven approaches: synchronization is easily obtained by event cooperation [10]. Another key advantage of events is scheduling at application level. Thus scheduling optimization is possible. Also events allow better code locality, which is one of implementation mechanisms of the fundamental principle of modern software development – Separation of Concerns (SoC) [11, 12, 13].

3.2 Event-based Specification Chart (ESC)

The main objective of research is to simplify the specification of concurrency in information system architecting. If events happen in component ports and relationships among events are the only things that interest us, an event-based approach for behavior specification can be an adequate choice.

Event-based Specification Charts (ESCs) consist in drawings, which specify events, event orderings, event conflicts, roles and role actions.

An *event* is an instantaneous, atomic "state" transition in the computation trace. Exactly over these transitions the behavior of a system is defined. Event ordering in computation trace is determined by the *causal dependency* relationship "$\rightarrow$" (read as "*precede*"). The interpretation of "$\rightarrow$" as a causal ordering means that, if $e1$ and $e2$ are events in a system and $e1 \rightarrow e2$, then existence of the event $e1$ will cause occurrence of the event $e2$ in the future.

Conflict relationship "$\leftrightarrow$" models mutual exclusion of events, disallowing them to overlap in time. An *asymmetric conflict* "$\rightarrowtail$", which blocks an event while another event happens, is allowed too.

Thus an event e can occur when all its causes have occurred and no event, which it is in conflict with, has already occurred.

Concurrency and *non-determinism* are implicit in ESC specifications, and are determined by causal independence and absence of conflicts between concurrent events.

A *role* defines the behavior of a participant, identified in the collaborations between architecture components. Through roles the component captures events, which can occur in an order determined by

concurrency and synchronization logic, and exposes actions associated to these events by *role to cloud incidence* connectors.

Possibility of localization of interrelated events is realized using *cloud*. The cloud of events is a partially ordered set of events, where the partial ordering is determined by causal, temporal and other relations between events. A closely related term of event cloud is defined in the glossary of *Event Processing Technical Society* too [14].

Basics of the language notations are presented in Figure 4, where ESC chart describes the following entities:

- Events: $e1$, $e2$, $e3$, $e4$, $e5$;
- Causal dependencies: $e1 \rightarrow e2$, $e1 \rightarrow e3$, $e4 \rightarrow e5$;
- Conflict: $e3 \leftrightarrow e4$;
- Clouds: *Cloud1, Cloud2;*
- Role: *Role1.*

In order to increase flexibility of specifications two relationship attributes for causal dependencies were introduced: *role incidence* and *event occurrence*.

Role incidence attribute is used to indicate which roles are implicated in event occurrence, and has the following values:

- *Same* – the causal and dependent events occur involving the same role;
- *All* – a special case of *same* attribute, used when a dependent event occurs only if the causal event is occurred in every incident role. Thus the cause event is synchronization between all component processes interfaced by roles.
- *Any* – a dependent event occurs, when the causal event manifested itself in any incident role; used as attribute for dependencies between events located in different clouds;

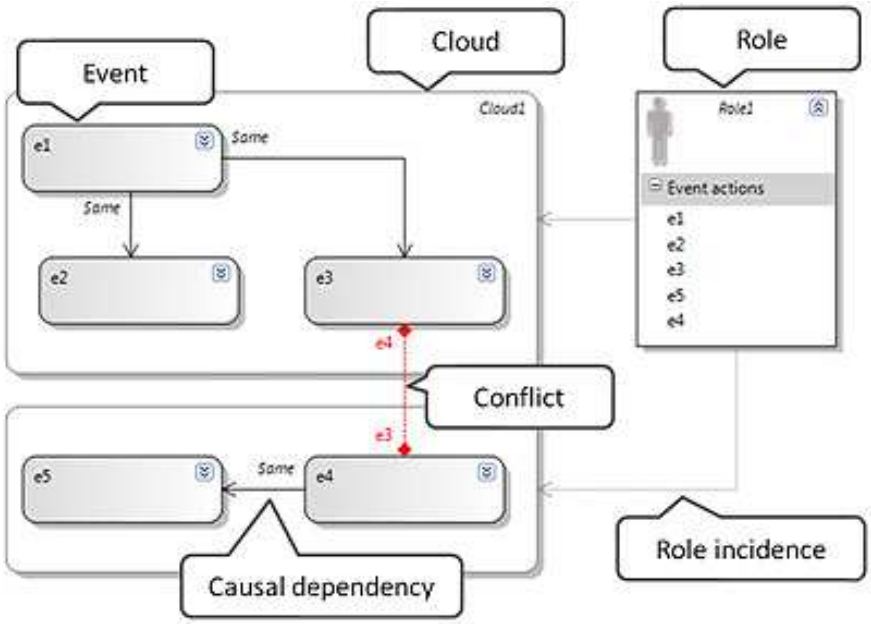

Figure 4. An ESC diagram

Event occurrence attribute can be used to specify how many times the dependent event may occur, when the causal event manifests once:

- *bound[n]* – limits to n number of occurrences of the dependent event in each incident roles;
- *free* – the dependent event may occur unlimited times for incident roles.

Let there be four events ordered by some dependencies ($e1 \rightarrow e2$, $e2 \rightarrow e3$, and $e2 \rightarrow e4$), which form a system behavior presented in Figure 5. *Role1* and *Role2* are synchronized by event $e1$, because dependency between $e1$ and $e2$ has *All* as value of *role incidence* attribute. Event $e2$ thus will not happen until $e1$ will not occur in both roles. Also according to diagram from Figure 5, the event $e3$ will occur only once

for each incident role (both *Role3* and *Role4*), but the event $e4$ will occur unlimited times involving incident roles.

It is important to mention that role can be "interpreted" by a process or multiple. Thus event $e3$ may occur once in multiple processes or will occur multiple times in one process (execution unit).

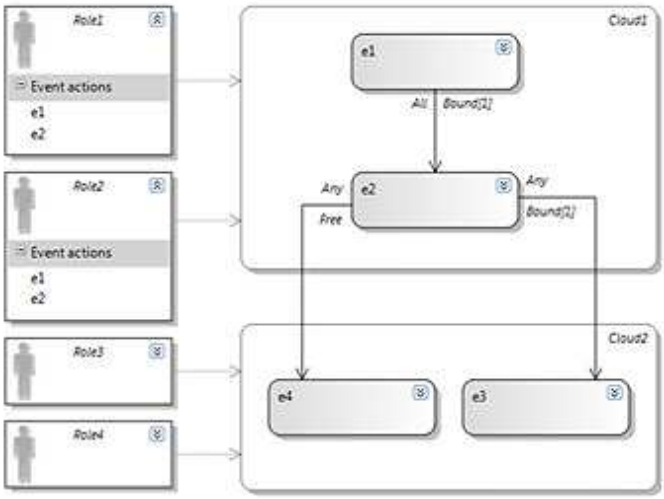

Figure 5. Synchronized events in ESC

Modern software development requires an incremental refinement approach. Thus reutilization is an important factor to efficient elaboration, but it is applied to late. Reutilization, usually achieved through inheritance, is especially present from design to programming activities. Therefore an effort must be made to apply reusing in behavioral specification of concurrency-intensive architecture too.

Refinements in ESC language are realized through *cloud superposition* (with *event specialization*) and *event substitution*.

Cloud superposition superimposes additional behavior on an existing cloud. Addition behavior preserves independencies and conflicts between old events, therefore *liveness* and *safety* properties will be preserved. Superposition must be viewed as *monotonic inheritance*,

because can be characterized in the following way: new events may be added; new actions may be associated to new events; new causal dependencies may be added between new events, new causal dependencies may be added between a specialized event and events from superpositioned cloud; a specialized event may be refined through *substitution* by events from a cloud.

Event substitution is a refinement, which replaces an event by multiple events from a cloud. Substitutions preserve dependencies of replaced events, therefore causal ordering is preserving too.

Refinement relations between clouds and events is shown in Figure 6, according to which *Role1* is involved by events $e1$, $e3$, $e4$, $e5$, $e7$ with occurring ordering determined by the following final (after refinements) causal dependencies: $e1 \rightarrow e3$, $e1 \rightarrow e4$, $e1 \rightarrow e5$, $e4 \rightarrow e7$, $e5 \rightarrow e7$.

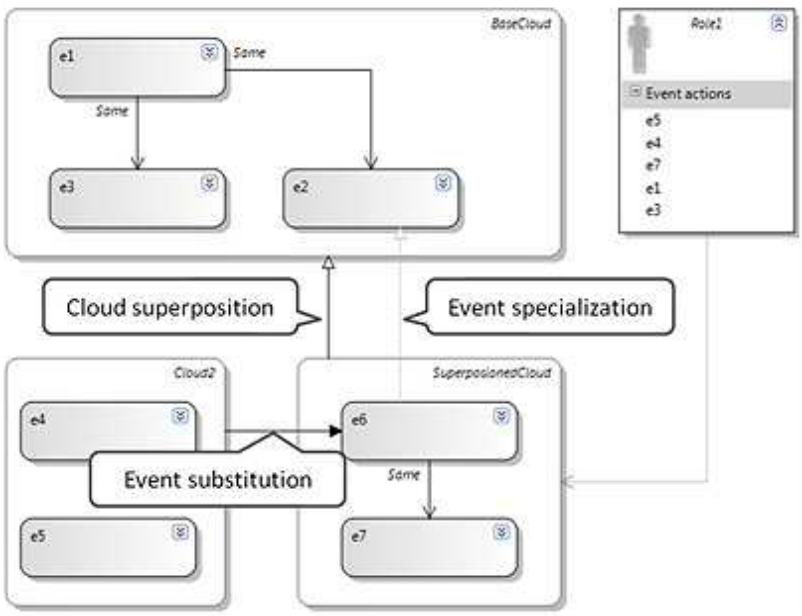

Figure 6. Superposition and substitution refinements

3.3 An event structure semantic for ESCs

Every specification language should have a formal semantic, which defines logic for reasoning about behavior and concurrency. A semantic is indispensable for the development of tools; therefore it is an advantage, if specification language has direct and simple correspondence to the logic. This is the way to avoid errors in concurrent specification.

Another important aspect of language design concerns explicit linking of specification to implementation, which is realized at later stages of developments. So language must be capable to preserve internal system "structure" (which can be lost with interleaving models [15]), to express implicitly non-determinism (which is an inherent property of modern multi-threading programs [16]), and to avoid detailed internal state description (behavioral approach is preferred for high-level architectural specification).

According to [15, 17] and aforementioned requirements, a well accepted branching-time true concurrency semantic model is model of event structures (Figure 7).

Figure 7. The ESC language as an emanation of branching-time true concurrency extensional semantic

An *event structures* model with non-interleaving semantic allows in

a natural way to describe the relationship between events of the system [15, 17].

The event structure $S = (E, \leq, \#)$ consists of a countable set of events E, partially ordered by causal dependence $\leq \ \subseteq (E \times E)$, an irreflexive and symmetric conflict $\# \subseteq (E \times E)$, satisfying the principle of inheritance: $\forall e, e1, e2 \in E;\ e\#e1 \leq e2 \Rightarrow e\#e2$. Thus the concurrency relationship *co* between events e and $e1$ from E can be defined as follows: e *co* $e1$, iff $\neg(e1 \leq e \vee e \leq e1 \vee e\#e1)$.

Event substitution refinement can be defined as *vertex substitution* operation presented in [18] with assumption that substituted event has no conflict relationship with any events (Figure 8).

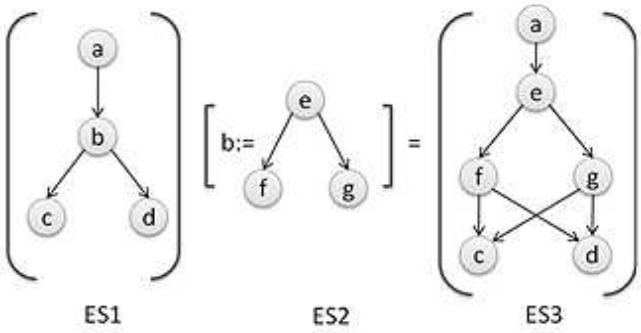

Figure 8. Event substitution

In order to describe formally this refinement operation it needs to assume that

- $U(E_1, e_1)$ is a subset of E_1 upper bounded by $e_1 \in E_1$ ($\forall u \in E_1$: $u \leq e_1$);
- $L(E_1, e_1)$ is a subset of E_1 lower bounded by $e_1 \in E_1$ ($\forall l \in E_1$: $e_1 \leq l$);
- $\lceil S \rceil$ denotes a subset of S, including only elements' upper bound S ($\forall s \in S;\ \forall u \in \lceil S \rceil;\ s \leq u$);

- $\lfloor S \rfloor$ denotes a subset of S, including only elements' lower bound S ($\forall s \in S;\ \forall l \in \lfloor S \rfloor;\ l \leq s$).

Then after substitution of event b in event structure ES1 by event structure ES2 (operation which can be noted as in [19] with ES3 = ES1[$b \rightarrow$ES2]) is obtained an event structure ES3=$(E_3, \leq_3, \#_3)$, where

- $E_3 = (E_1 \cup E_2) - \{b\},\ E_1 \cap E_2 = \emptyset$;
- $\leq_3 = (\leq_1 - \leq_b) \cup \leq_2 \cup \leq_{\lceil U(E1,b) \rceil \times \lfloor E2 \rfloor} \cup \leq_{\lfloor L(E1,b) \rfloor \times \lceil E2 \rceil}$;
- $\#_3 = \#_1 \cup \#_2$.

Refinement through *cloud superposition* formally can be interpreted as union of two graphs, which anticipates vertex contractions specified by specialization relationships (dashed arrow in Figure 9).

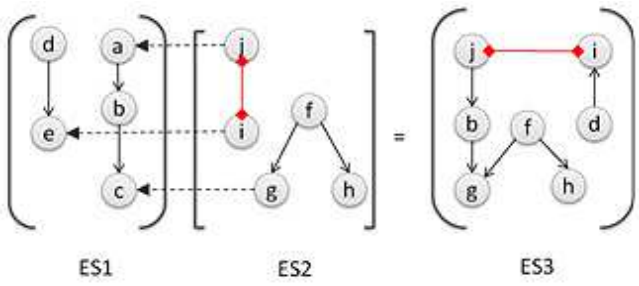

Figure 9. Cloud superposition

The above-described basic relationships are the ones which permit event structures to be expressive and natural for concurrency description and explaining. However these arguments are not valid for automated analysis, because of the difficulty in defining operations like products and parallel compositions on event structures of the form $(E \leq, \#)$. They encourage the uses of more general structures or even CCS/CSP like languages from which event structure semantics is then induced [15].

3.4 Operational interpretation of ESC specifications

Automated analyses imply an operational interpretation of formal ESC specifications. The most appropriate for this is the classic process algebra CSP (*Communicating Sequential Processes*). CSP has an event-based operational semantic and describes system behavior by sequencing of events [20]. There are two *popular* model checkers based on CSP-like operational semantic and syntax: FDR (Failures/Divergence Refinement), which is a commercial product expressing models in machine-readable dialect of CSP (CSP_M) [21]; and PAT (Process Analysis Toolkit) with CSP# as input language [22].

Greater flexibility, expressiveness, openness and freeness offered by PAT determine a univocal choice for CSP# as interpretation language. Compared to CSP_M, CSP# adds to original language features such as shared variables, asynchronous communication channels and event associated programs [23].

Formally, the language of sequential processes (programs) CSP#, ranged over processes P and Q, is the set of terms generated by the following BNF *(Backus-Naur Form)* description (Figure 10), in which e is a name representing an event with an optional sequential program prog, X is a set of event names, b is a Boolean expression,ch is a channel, exp is an expression, and x is a variable.

```
P ::= Stop | Skip                                          (event prefixing)
  | e{prog} -> P                                           (channel operations)
  | ch!exp -> P | ch?x -> P                                (hiding event and interleaving)
  | P \ X | P ||| Q                                        (sequential and parallel composition)
  | P;Q | P || Q                                           (external and internal choices)
  | P[]Q | P<>Q                                            (conditional choice)
  | if b {P} else {Q}                                      (guarded process)
  | [b]P
  | P interrupt Q
```

Figure 10. The BNF description of CSP# language

The easiest way to apply PAT platform as model checker is to create a "rewriter" from the ESC specification language to CSP# language. In

order to avoid complex CSP# models, rewriting should be anticipated by refinement operations, which can be understood as operations of syntactic substitution.

Rewriting from ESC model to CSP# model produces systems composed of multiple *processes*. In accordance with CSP# language definition the word *process* can stand for the behavior pattern of an object, which can be described in terms of limited set of events. In both languages each event name denotes an event class; there may be many occurrences of the same event, involved separately in time by processes (CSP#) or roles (ESC).

Let *Role1* and *Role2* be the roles, which act and interact with each other exactly in accordance with ESC specifications showed in Figure 11. It is easy to see, that *Role1* exposes actions for occurrences of events a, b and c, ordered by causal dependencies from *TheCloud1*; and the behavior of *Role2* is determined by ordering of occurrences of events a, b, c (from *TheCloud1)*, and d, e, f (from *TheCloud2*). Also according to specification, event b from *TheCloud1* requires simultaneous participation of the both roles involved.

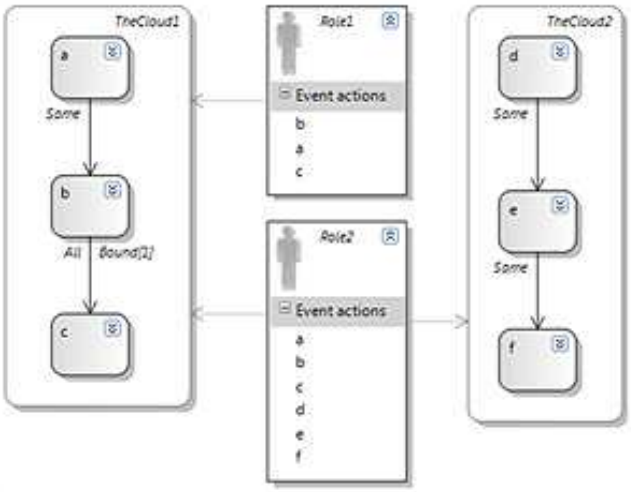

Figure 11. Event clouds as behavior patterns

The system behavior is defined by all role behaviors, which can be described as combination of deterministic CSP processes. When roles are incident to the same cloud, parallel composition is used. *Parallel composition* introduces concurrency and synchronization. Using of the same alphabet assures that processes interact in lock-step synchronization over events with the same name. *Interleaving* operator may combine many role behaviors (incident to different event cloud) into complex one, representing completely concurrent activity.

Thus a CSP# model which describes a behavior equivalent to ESC specification from Figure 11 consists from five processes (Figure 12): a composed process *System*, which corresponds to overall specified system; processes $R1$ and $R2$, which correspond to roles *Role1* and *Role2*; processes $R21$ and $R22$, which combine concurrently behavior of *Role2* determined by *TheCloud1* and *TheCloud2* from ESC specification.

```
R1 = a1 -> b -> c1 -> R1;
R21 = a2 -> b -> c2 -> R21;
R22 = d2 -> e2 -> f2 -> R22;
R2 = R21 ||| R22;

System = R1 || R2;
```

Figure 12. Interleaving and parallel composition in CSP# model

Also parallel composition is used to define a behavior like presented in Figure 13, according to which *Role1* may concurrently involve events (b and d, b and e, c and e).

After model "rewriting", the role process $R1$ behaves like the system composed of processes $R11$, $R12$ and $R13$, interacting in lock-step synchronization as described above (Figure 14).

Last translations may not be optimal, because the processes describing the behavior *Role1* are obtained by finding in the cloud's DAG all the paths that begin from source events and end at the sink events. The equivalence of ESC model and CSP# model can be proved, if both models satisfy the same state graph (Figure 15). This equivalence is

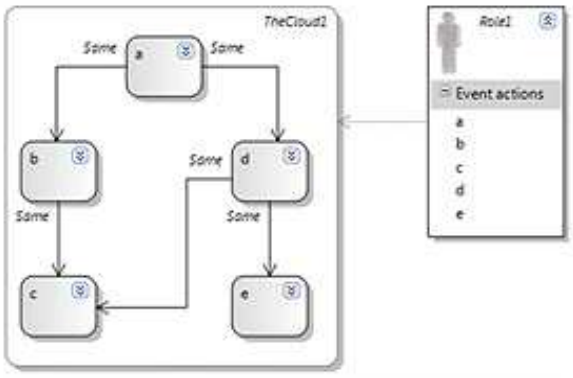

Figure 13. A cloud with concurrent events

```
R11 = a -> b -> c -> R11;
R12 = a -> d -> c -> R12;
R13 = a -> d -> e -> R13;

R1 = R11 || R12 || R13;
```

Figure 14. One cloud can impose multiple processes for behavior definition

analogical to Milner's principle of observational congruence [20].

It is easy to see from the state graph (Figure 15) that two independent events (c and e, d and b, b and e) which can occur at the same state, can be performed in any order without affecting the reached state (in literature this is often identified as 'diamonds' of concurrency [17]).

In Figure 16 represented role behaviors are defined by two clouds, which include conflicting events. Conflict relationship is used to specify mutual exclusion between events. Inheritance principle of conflict relationship acts in cloud boundary. Circle in the upper-left corner of cloud shape means that the cloud is a singleton: at the same time only one role incident to cloud can involve events from this cloud.

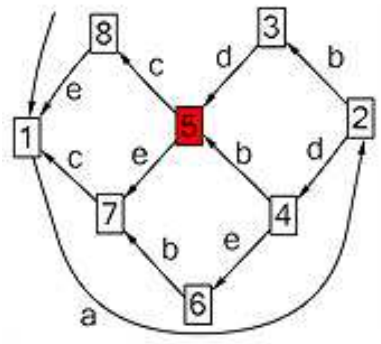

Figure 15. Complete state graph (generated by PAT)

It is obvious that conflict relationship imposes a choice between processes including conflicting events. Selection in CSP model will be done in the *conflict process* specifically designed for this. Conflict process must be defined by role events. Deterministic choice realized in CSP by a special operator – *external choice,* can help us to engage in execution only one "conflicting" process. Thus specification shown in Figure 16 can be translated as presented in Figure 17.

In the system described in Figure 17, the lock-step synchronization admits execution of one process at a moment. Using global variables and guarded processes, and splitting alternative process in two (sub) processes can permit occurrence of one conflicting event in multiple role processes (Figure 18).

4 Conclusion

The Event-based Specification Charts (ESC) specification language was created to easily design model of concurrent processes using events. Thus, the proposed language involving event-driven techniques, does not replace the existing verification methods and tools. On the contrary, the main objective of the language is to improve usage of formal methods in verification.

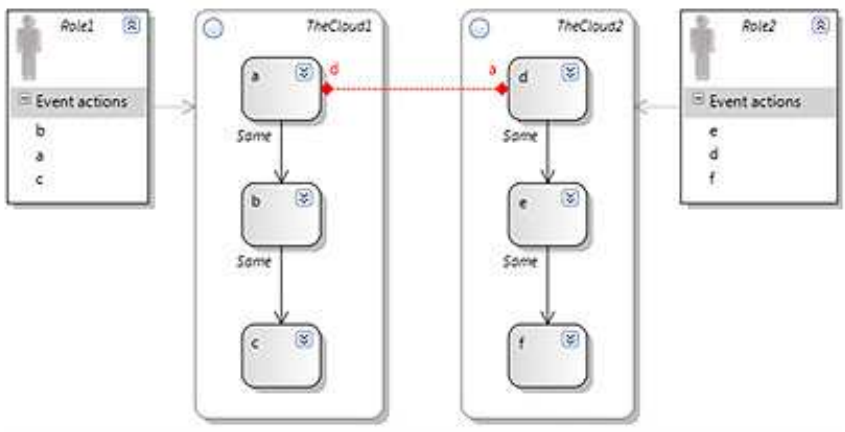

Figure 16. Conflict between events from singleton clouds

```
R1 = a -> b -> c -> R1;
R2 = d -> e -> f -> R2;
Conflict = a -> c -> Conflict [*] d -> f -> Conflict;

System = Conflict || (R1 ||| R2);
```

Figure 17. Conflict relationship and external choice

Event-based specification charts have many advantages. They are easily understood, even by the non-specialist. They permit to concentrate on causality or independence of events. They offer reutilization mechanisms of events.

Language constructs semantically are interpreted by means of event structure model, which is more suitable for modeling "true parallelism". Hoare's CSP language provides operational interpretation for ESC models: partial order is modeled by the sequencing; conflict relation is modeled by nondeterministic choice, etc.

Major factor reducing the use of formal language is poor integration of formal tools with modern development environments. Now we intend to develop a tool for specification of concurrency, integrated into Visual

```
Var cr1 = 0;
var cr2 = 0;

R1() = a -> b -> c -> R1();
R2() = d -> e -> f -> R2();
Cf() = [cr2 == 0] a -> {cr1++} -> Cf() []
   [cr1 > 0] c -> {cr1--} -> Cf() []
   [cr1 == 0] d -> {cr2++} -> Cf() []
   [cr2 > 0] f -> {cr2--} -> Cf();

System() = Cf() || (|||x:{1..2}@(R1() ||| R2()));
```

Figure 18. Multiple role processes with conflicting events

Studio IDE, using Visual Studio Visualization and Modeling SDK, in order to increase the quality of software development. Formal checks will be made by PAT, which provides an API just for this.

References

[1] D. Ciorba, V. Besliu. *Architecting software concurrency.* Computer Science Journal of Moldova. Chisinau : s.n., 2011. Vol. 19, 1(55), pp. 92–108. http://www.math.md/files/csjm/v19-n1/v19-n1-(pp92-108).pdf. ISSN 1561-4042.

[2] M. Heisel. *A pragmatic approach to formal specification.* [ed.] H. Kilov and W. Harvey. Object-oriented behavioral specification. Boston / Dordrecht / London : Kluwer Academic Publishers, 2006.

[3] A. Hall. *Realising the Benefits of Formal Methods.* Journal of Universal Computer Science. 2007, Vol. 13, 5.

[4] LKP Consulting Group. *The Real Cost of Software Defects.* [http://www.lkpgroup.com/Cost%20of%20Software%20Defects.pdf] Atlanta, US : s.n., May 2006.

[5] Intel Corporation. *Intel® Fast Track Initiative: New Tools for Enabling the Latest Technologies.* [http://www.intel.com/partner/ft_webinar/Intel-FastTrack-SAT-Whitepaper_FINAL_4-14-2010.pdf] 2010.

[6] Object Management Group Inc. *UML 2.3. The Object Management Group.* [Online] May 9, 2010. [Cited: Nov 21, 2010.] http://www.omg.org/spec/UML/2.3/.

[7] Object Management Group Inc. *Object Constraint Language (OCL).* [Online] Feb 2010. [Cited: Nov 21, 2010.] http://www.omg.org/spec/OCL/.

[8] A. Kompanek. *Modeling a System with Acme.* [http://www.cs.cmu.edu/∼acme/html/WORKING-%20Modeling%20a%20System%20with%20Acme.html] s.l. : Carnegie Mellon University, 1998.

[9] M. Welsh. *An Architecture for Highly Concurrent, Well-Conditioned Internet Services.* Ph.D. Thesis. Berkeley : University of California, August 2002.

[10] M. Fowler. *Event Collaboration.* Development of Further Patterns of Enterprise Application Architecture. [Online] Jun 19, 2006. [Cited: Dec 05, 2010.] http://martinfowler.com/eaaDev/EventCollaboration.html.

[11] W. Hürsch, C. Lopes. *Separation of Concerns.* College of Computer Science, Northeastern University. Boston, USA : s.n., 1995. Technical report.

[12] C. Constantinides, T. Elrad. *On the Requirements for Concurrent Software Architectures to Support Advanced Separation of Concerns.* OOPSLA'2000, Workshop on Advanced Separation of Concerns in Object-Oriented Systems. 2000.

[13] G. Kiczales, et al. *Aspect-Oriented Programming.* Proceedings of ECOOP'97. s.l.: Springer-Verlag, 1997.

[14] Event Processing Technical Society. *Event Processing Glossary.* [http://www.ep-ts.com/] [ed.] D. Luckham and R. Schulte. July 2008.

[15] G. Winskel, M., Nielsen. *Models for Concurrency.* [http://www.daimi.au.dk/PB/463/PB-463.pdf] November 1993. DAIMI PB-463.

[16] R. Carver, K. Tai. *Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs.* [http://books.google.ro/books?id=Wuex4orZgOEC] s.l.: John Wiley and Son, 2006. ISBN 0-471-72504-8.

[17] V. Sassone, M. Nielsen, G, Winskel. *Models for Concurrency: Towards a Classification.* Theoretical Computer Science. 1996. Vol. 170, 1-2, pp. 297–348.

[18] T. Basten. *Parsing Partially Ordered Multisets.* International Journal of Foundations of Computer Science. s.l. : World Scientific Publishing Company, December 1997. Vol. 8, 4, pp. 379–407.

[19] U. Goltz, R. Gorrieri, A. Rensink. *Comparing Syntactic and Semantic Action Refinement.* Information and computation. s.l.: Academic Press, Inc., 1996. 125, pp. 118–143.

[20] C. Hoare. *Communicating Sequential Processes.* [http://www.usingcsp.com/] s.l.: Prentice Hall International, 2004.

[21] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement. FDR2 User Manual.* [http://www.fsel.com/fdr2_manual.html]. Octomber 2010.

[22] Y. Liu, J. Sun, J. Dong. *Developing Model Checkers Using PAT.* [ed.] A. Bouajjani and W. Chin. Automated Technology for Verification and Analysis – 8th International Symposium, ATVA 2010. Singapore : Springer, September 2010. Vol. 6252, pp. 371–377.

[23] J. Sun, et al. *Integrating Specification and Programs for System Modeling and Verification.* [ed.] W. Chin and S. Qin. Proceedings of the third IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09). s.l.: IEEE Computer Society, 2009. pp. 127–135.

Dumitru Ciorbă, Victor Beşliu

Received May 28, 2011

Technical University of Moldova
Automation and Information Technology Department
Str. Studenţilor, 7/3, corp 3, 504 Chişinău, MD-2068
Phone: (+373 22) 509908
E–mail: *victor.besliu@ati.utm.md*, *besliu@mail.utm.md*, *vbesliu@yahoo.com*
E–mail: *diciorba@yahoo.com*, *dumitru.ciorba@ati.utm.md*