Electronic Communications of the EASST
Volume 14 (2008)

Proceedings of the Third Workshop on Petri Nets and Graph Transformations (PNGT 2008)

Debugging Distributed Systems with Causal Nets

Gian Luigi Ferrari, Roberto Guanciale, Daniele Strollo and Emilio Tuosto

10 pages

Guest Editors: Paolo Baldan, Barbara König
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Debugging Distributed Systems with Causal Nets

Gian Luigi Ferrari[1], Roberto Guanciale[2], Daniele Strollo[34] and Emilio Tuosto[5]

[1] giangi@di.unipi.it
[3] strollo@di.unipi.it
Dipartimento di Informatica
Università degli Studi di Pisa, Italy

[2] roberto.guanciale@imtlucca.it
[4] daniele.strollo@imtlucca.it
Institute for Advanced Studies
IMT Lucca, Italy

[5] et52@mcs.le.ac.uk
Computer Science Department
University of Leicester, UK

Abstract: Formal methods for deciding the properties of service oriented systems are of paramount importance. However, they may require to master sophisticated techniques that programmers may lack. This issue can be mitigated by providing programmers with tools and techniques that are close to the usual programming practice. Here, we propose to use causal nets to define a few debugging primitives to drive the analysis of system developed with SC (after Signal Calculus), a process calculus featuring event-notification communication. The usage of causal nets permits, transparently to the programmer, to build up the causal information of systems along their evolution. A few debugging primitives can be defined in terms of operations on the causal nets. Using the debugging primitives, programmers can query the causal nets in order to reconstruct how the computation evolved. [1]

Keywords: Causal nets, event notification, debugging

1 Introduction

Modern computing is typically distributed as data are stored on systems that are connected to form huge networks and computations take place on several locations. Moreover, emergent metaphors for programming modern networks (aka *overlay computers*) aim to dynamic composition of distributed computational units. For instance, the *service oriented computing* paradigm promotes the composition of distributed services (the unit of computation) that are available and

[1] Research supported by the EU FET-GC2 IST-2004-16004 Integrated Project SENSORIA and by the Italian FIRB Project TOCAI.IT.

can be dynamically bound and interact. Formal methods for deciding (and measuring) the properties of such systems are of course of paramount importance. However, they may require to master sophisticated techniques that programmers may lack. This issue can be mitigated by providing programmers with tools and techniques that are close to the usual programming practice. Following this approach, in [5] causal nets have been used to define a few debugging primitives that may help in the analysis of mobile code for the ambient calculus [1]. Figure 1 gives a pic-

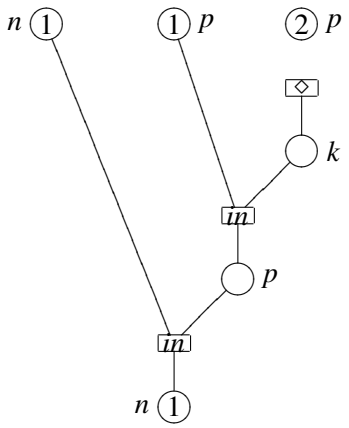

Figure 1: A causal net for the ambient calculus

torial representation of a trace of an ambient process in terms of causal nets. Places are labelled by either ambient names (if any) or by $\circ$ and denote the states of ambient processes (places carrying the same ambient names are told apart from indexes). Transitions are labelled by ambient capabilities (*in*, *out*, *open*) or by *sync* or $\diamond$ (representing communication and the arrival (resp. departure) of an external (resp. internal) ambient). For instance, the ambient k in Figure 1 enters p after exiting its surrounding ambient.

A key idea of [5] is the usage of causal nets that, transparently to the programmer, build up the causal information of mobile ambients along their evolution. In fact, in [5] a few debugging primitives are defined in terms of operations on the causal nets associated with ambients in order to trace their migrations/synchronization along the computation. Using the debugging primitives, programmers can "query" the causal nets in order to reconstruct how the computation evolved. For instance, the debugging capabilities allow one to ask for the (sub-)causal net involving the ambient name p in the net of Figure 1 (which will return the whole net but the lowermost *in* transition and its n-labelled places). In this way programmers can interweave debugging primitives with ambient capabilities so that information about the current computation can be acquired.

This short paper drafts a few embryonic research ideas by combining the preliminary ideas presented in [5] and more recent research directions emerging in the context of the SENSORIA and TOCAI projects.

2 Basics of JSCL and SC

Our research program proposes to extend the approach presented in [5] to the JSCL (after Java Signal Core Layer) language [2] inspired by SC (after signal calculus) [3], a process calculus featuring event-notification communication mechanism for coordinating distributed component. Indeed, we contend that JSCL and its underlying theoretical model SC are more suitable and effective than the ambient calculus for extending and applying the ideas in [5].

We first give a hint of JSCL so that we can ground our contention on its suitability as a model for a distributed debugging framework. Details on JSCL implementation can be found in [2] while its SC model is described in [3].

A JSCL system consists of *components*, possibly allocated on different execution sites and joining other running components. Components interact using *signals* classified according to *topics* and are delivered on *flows* and trigger *reactions*. Flows route signals to components while reactions basically are signal handlers: the process associated with a reaction is executed when a signal of the same topic as the reaction reaches a component. The communication of events is performed in two phases. Initially, once a component raises an event, it spawns several envelopes, each of them targeted to one of the subscribers. Asynchronously, each subscriber will be able to retrieve its pending envelopes and consume them. Figure 2 pictorially shows two JSCL components `P` and `C` (after producer and consumer respectively) connected by two flows for topics `t` and $\texttt{t}'$ from `P` to `C` trough which signals of different topics can be delivered to the consumer in order to be processed. Upon emission of a signal of a topic from `P`, say `t`, the corresponding

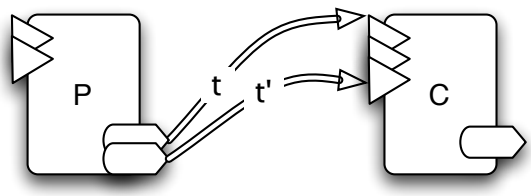

Figure 2: Producer-consumer in JSCL

reaction (graphically represented as a triangle in Figure 2) of `C` is triggered so that a process will start its execution within the component `C`. Notice that different flows trigger different reactions; in fact, a signal of topic $\texttt{t}'$ will trigger a different reaction.

Flows and reactions can be dynamically added to components and a signal is received by *all* the components for which the emitter components has a flow typed with the same type as the signal. For instance, the producer in Figure 2 can add a `t` flow towards another component, say $\texttt{C}'$, so that each signal of topic `t` would be delivered to both `C` and $\texttt{C}'$.

3 A Debugging Model for SC and JSCL

We argue that causal nets can be used more suitably in the context of SC/JSCL. One of the main drawbacks of the approach in [5] lays on the fact that programmers are required to mix debugging

primitives in their code. Albeit not having any impact on the semantics of programs[2], this is not ideal as it forces programmers to reason on which region of the code debugging primitives must be put which is typically hard in a distributed setting due to the high degree of non-determinism usually present in such systems. This problem can be tackled by exploiting the event-notification featured by SC and supported by JSCL for coordinating distributed components. If fact, one can use special components, called *debunits* (after debugging units), that trigger debugging processes when some events occur. More precisely, a debunit can react to events as any other component but, instead of executing a normal program, it runs e.g. queries on the causal nets of some components so that the programmer can be reported on the states of the computation (s)he is interested in.

As an example, consider Figure 3 where a debunit is added to the example in Figure 2. Components `P` and `C` run in "debug mode" (graphically represented by the nets drawn in their

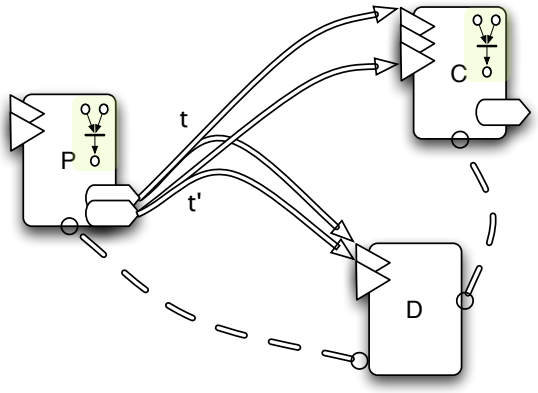

Figure 3: Debugging SC and JSCL

boxes), namely they log their execution and build the causal net yielding the history of their computation. The dashed lines connecting the debug unit `D` and the components `P` and `C` represent special connections described below. Notice that the debunit `D` in Figure 3 has reactions for events of both topics `t` and $\mathtt{t}'$; however, it is not required that debunits react to signals of all topics. Figure 3 relies on the following assumptions:

1. components can interact with debunits through special channels (the dashed lines in Figure 3) whereby causal nets can be communicated to debunits;

2. the code associated with reactions of debunits performs debugging operations on the causal nets obtained by the components;

3. debunits do not issue events for other components, namely debunits do not have flows towards normal components (as `D` in Figure 3) but they can use flows to coordinate among them.

[2] The debugging primitives only act on the causal nets logging the computation.

Assumption 1 can be discharged as the communication of causal nets among components and debunits can be encoded in SC and therefore implemented in JSCL. However, this interaction is transparent to the programmer and therefore its adoption simplifies the presentation of debugging primitives.

Assumption 2 is made for the sake of simplicity of this germinal version. Indeed, in this way debunits become "passive" observers, namely they can only acquire information produced by normal components but cannot generate events. We think that it would be useful and worth exploring more sophisticated debugging mechanism requiring "active" debunits that allow them to fully exploit the event-notification primitives of the SC/JSCL framework. For instance, desirable debugging mechanisms would be to

- allow debunits to temporarily inhibit/enable selected reactions or flows of components;
- add special reactions to components triggered by "debugging" topics in order to constrain the flow of control (as much like *breakpoints* do in typical debuggers);
- issue events carrying particular data as typically done during the testing phase of applications.

Our plan is to design the debugger in SC and integrate it in the JSCL Java platform in such a way that those features can be smoothly added at the later stage. It is very likely that the event-notification based coordination of our framework can make such kind of extensions straightforwardly achievable.

An aspect worth remarking is that debunits avoid debugging primitives to be scattered in the code. In the proposed model indeed debunits and components interact by exploiting the event-notification paradigm featured by our framework.

Last but not least, it should be noted that JSCL can be envisaged as a Java API (implementing SC), therefore its usage does not require theoretical background and, more interestingly, it is amenable to be integrated with the Java Platform Debugger Architecture (JPDA). We envisage this as one of the strong elements of our proposal: debugging primitives are formally defined using causal nets and can be given a precise semantics in the context of SC; however, this is transparent to programmers who can simply use the debugging primitives as much as they do in most of the available debuggers. At the same time practitioners can take advantage of both the debugger for distributed component coordination as well as of JPDA to debug computations local to each component and mostly concerning with the Java code not related to distributed coordination.

4 Debugging Sessions

The main ingredients of causal networks are places, that represent the configurations, and transitions, that represent the actions of interest that have been performed. The representation of SC computations in the causal networks requires to isolate the set of actions that are relevant for the debugging feature we want to expose and the kind of configurations that can be built on them. A distinguishing feature of SC model relies on the event notification paradigm supporting asynchronous communications. Hence, relevant actions are considered the rising of an

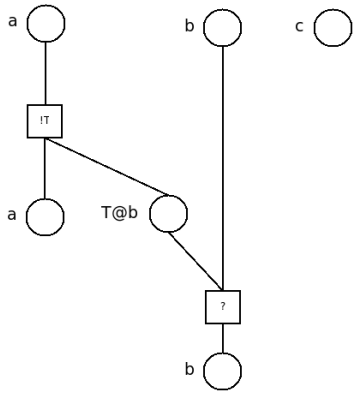

Figure 4: Asynchronous notification

event (denoted by "$!T$") and the activation of a reaction (denoted by "?"). For this reason further activities modeled in SC, such as the installation of a new reaction and the modification of the subscriptions (the managing of the flows), are not explicitly reflected in the structure of our causal networks. The places are used to keep track of the components defined in SC and the set of pending envelopes (denoted by "$T@a$"). Roughly speaking, this kind of networks permits to represent computations in the asynchronous method invocation fashion.

In the following we explain by examples how the causal networks can be built by the SC run-time. We remark that the SC semantics has been formally described in [3].

4.1 Asynchronous notification

Our first example involves the asynchronous notification among components, the causal network built during the execution is reported in Figure 4.

The following SC term describes three components a, b and c.

$$a[\mathtt{out}\langle T\rangle]^0_{T:b} \parallel b[0]^{T:B}_0 \parallel c[0]^0_0$$

The three places a, b and c at the top of the causal network reflect this initial configuration.

The components b and c are both "inactive", meaning that they are not performing any action that can cause transitions of the system. The component a is "active", since it can notify ($\mathtt{out}\langle T\rangle$) the occurrence of a new event having topic T. The component searches into its flow ($T : b$) the set of subscribers for the raised event and then spawns into the network an envelope ($\langle T\rangle@b$) for each of them. Hence, the system evolves into the new one:

$$a[0]^0_{T:b} \parallel b[0]^{T:B}_0 \parallel c[0]^0_0 \parallel \langle T\rangle@b$$

As consequence, the run-time connects, into the causal network, the initial place a with a new transition "$!T$", representing the raising of the event. Moreover, the resulting places of the transition are a and $T@b$, stating that the component a does not disappear and that a new envelope becomes visible.

Being b capable of consuming this notification, since it has a suitable reaction ($T : B$), it internally activates the proper behavior.

$$a[0]^0_{T:b} \parallel b[B]^{T:B}_0 \parallel c[0]^0_0$$

The run-tine connects the places representing the reacting component and the one representing the envelope with a new transition "?". Moreover, the resulting place of the transition is only b, representing that the envelope has been consumed.

An additional consideration is needed. Even if components flows are not explicitly modeled in our causal networks, this does not impact with the observable actions of the debugger. In fact, the flow configuration of a component can be inferred by observing the set of places generated by the emission transition.

4.2 Anonymity of envelopes

A key aspect of SC is that the envelopes, the asynchronous notification of events, are anonymous. Namely, if two identical events are notified to a component, the developer should not be interested in which one is consumed during the activation of the reaction. The following example describes a system in which the same event is notified twice:

$$a[\texttt{out}\langle T\rangle;\texttt{fupd}\,(T : c);\texttt{out}\langle T\rangle]^0_{T:b} \parallel b[0]^{T:B}_{F_b} \parallel c[0]^0_{F_c}$$

The causal network corresponding to this configuration has three initial places, representing the three components. Initially, the system performs the same action described in the previous example, spawning the envelope targeted to the component b.

$$a[\texttt{fupd}\,(T : c);\texttt{out}\langle T\rangle]^0_{T:b} \parallel b[0]^{T:B}_{F_b} \parallel c[0]^0_{F_c} \parallel \langle T\rangle@b$$

As described above, the rising of an event is represented in the causal network by appending a transition having a place for each generated envelope. After the emission of the envelope, the component a updates its flow ($\texttt{fupd}\,(T : c)$), adding the component c to the subscribers of events T. Since flow updates are not traced at run-time, the causal network does not change during this execution.

$$a[\texttt{out}\langle T\rangle]^0_{T:b,c} \parallel b[0]^{T:B}_{F_b} \parallel c[0]^0_{F_c} \parallel \langle T\rangle@b$$

Referring to Figure 5 and Figure 6, two possible traces are possible depending on whether b consumes the pending envelope before the second notification of a. The causal network of system depicted in Figure 5 corresponds to the following SC coding:

$$a[\texttt{out}\langle T\rangle]^0_{T:b,c} \parallel b[B]^{T:B}_{F_b} \parallel c[0]^0_{F_c}$$

The behavior B is internally executed and the envelope consumed. The causal network is updated as described in the previous example. Finally, the component a raises again a T event, generating the proper envelopes.

$$a[0]^0_{T:b,c} \parallel b[B]^{T:B}_{F_b} \parallel c[0]^0_{F_c} \parallel \langle T\rangle@b \parallel \langle T\rangle@c$$

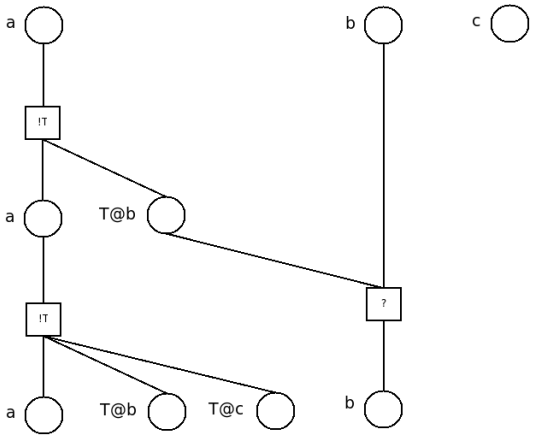

Figure 5: Early activation

Having a new flow structure, the component spawns an additional envelope for the new subscribed component. Notice that the changes applied to the flow of a, can be inferred in the causal net by the presence of the new place $T@c$.

If the second notification is performed by a before b activates its reaction, the system evolve as follows:

$$a[\mathtt{out}\langle T\rangle]^0_{T:b,c} \parallel b[0]^{T:B}_{F_b} \parallel c[0]^0_{F_c} \parallel \langle T\rangle@b \parallel \langle T\rangle@b \parallel \langle T\rangle@c$$

The two pending envelopes targeted to b could not be distinguished, being identical. One of them is consumed for the execution of the reaction of b

$$a[0]^0_{T:b,c} \parallel b[B]^{T:B}_{F_b} \parallel c[0]^0_{F_c} \parallel \langle T\rangle@b \parallel \langle T\rangle@c$$

When several identical envelopes are pending it should be interesting to count their instances. The way of implementing this feature is represented by the transition for the reaction activation in Figure 6. Let n be the number of places representing the pending envelopes, the transition consumes all these places and appends $n-1$ places corresponding to envelopes that are not been consumed.

5 A Research Plan

The main research activities necessary to deliver the model described in § 3 can be split in the following tasks:

1. definition of causal nets;
2. definition of SC-related (passive) debunits;
3. JSCL implementation and assessment of primitives;

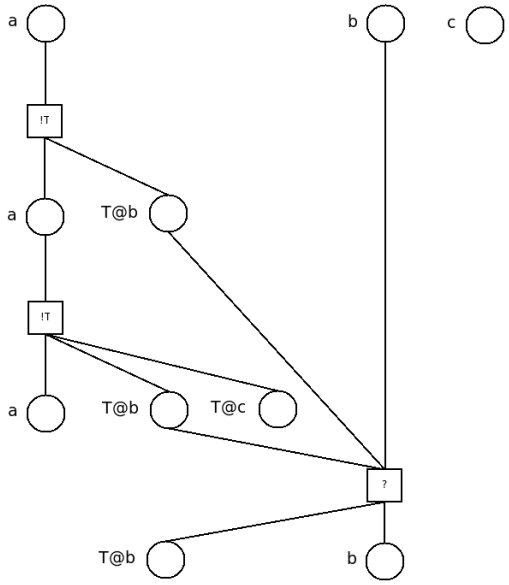

Figure 6: Late activation

4. definition of active debunits and their integration.

We argue that task 1 can be steadily adapted from [5] where causal nets for the ambient calculus have been specified in a way that is largely independent from the calculus and we think we can cast them easily to JSCL and SC.

Task 2 is instead more complex and delicate as it requires to individuate which primitives are desirable/useful in the context of JSCL and SC. This is an interesting research direction to explore because it has to deal with the trade-off between expressiveness and simplicity. We indeed contend that though viable, some sophisticated mechanisms should be avoided if they introduce complexity in the model that may affect usability of the language. Typical primitives that should be provided are the selection of the set of topics that must be traced and a query language for causal networks. This language should permit to activate the debunits if a specific configuration or a particular transition have been reached.

Task 3 is also crucial for our goal. In fact, the integration of the theory in a usable language is decisive for assessing the effectiveness of the proposed primitives and their usability. Given that JSCL faithfully implements SC, we argue that extending the language with the debugging primitives should be done by following to main principles: modularity and transparency. By modularity we mean that the implementation should take into account the possible evolutions of the debugging language due to e.g., task 4. Transparency is intended to hide as much theoretical details as possible from the programmer. Eventually, we would like to have debugging language that does not require the programmer to deal with causal nets or SC.

Task 4 has also a paramount importance. The debugging language should allow tight control when the programmer needs it. For instance, being able to tune reactions and flows responsiveness is a key feature.

We aim to a definition of active debunits that avoids scattering debugging primitives in the code. At a first glance, it seems that active debunits require control mechanisms that must be dispersed in the code of components. However, we argue that this can avoided by associating flows and reactions with debug modalities that can enable/disable them.

Finally, in recent work [4], we have proposed a few refactoring rules for distributed transactions implemented in SC/JSCL. The refactoring rules are basically graph transformation rules[3] that preserve weak bisimulation and are designed at the SC level. The design of such transformations must be done very carefully as it is easy to define appealing rules that spoil weak bisimulation and hence do not preserve the intended semantics. We think that a useful benchmark for the debugging language is its use within this contexts. In fact, it is very likely that when a transformation rule is not bisimulation preserving, the debugger may help in finding executions that wrecks the bisimulation and amend the rule to regain it.

Bibliography

[1] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240, 2000.

[2] GianLuigi Ferrari, Roberto Guanciale, and Daniele Strollo. Jscl: A middleware for service coordination. In Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge, editors, *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2006.

[3] GianLuigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. Coordination via types in an event-based framework. In John Derrick and Jüri Vain, editors, *FORTE*, volume 4574 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2007.

[4] GianLuigi Ferrari, Roberto Guanciale, Daniele Strollo, and Emilio Tuosto. Refactoring long running transactions. *WS-FM 2008 - 5th International Workshop on Web Services and Formal Methods*, 2008. To appear.

[5] GianLuigi Ferrari and Emilio Tuosto. A Debugging Calculus for Mobile Ambients. In *ACM Symposium on Applied computing*, Las Vegas (Nevada USA), March 11-14, 2001. ACM Press.

[3] This rules transform the graphs of SC/JSCL networks as those given in Figures 2 and 3.