# On Practical Verification of Processes

Rick van Rein
vanrein@cs.utwente.nl

University of Twente, Enschede, The Netherlands

**Abstract.** The integration of a formal process theory with a practically usable notation is not straightforward, but it is necessary for practical verification of process specifications. Given such an intermediate language, a verification process that gives useful feedback is not trivial either: Model checkers are not powerful enough to deal with object models, and theorem provers provide insufficient feedback and are not certain to find a proof.
Our work on life cycles has provided us with a precise process notation which is expressive enough to capture customary object designs. In this position paper we present our plans to perform verification on life cycles in such a way that it can provide useful feedback and at the same time proves a reasonable class of systems correct.

## 1 Introduction to Life Cycles

In this section we shall introduce our process language, called *Life Cycles*. Consider the diagram in Fig. 1. This is a life cycle, where transitions selectively cause transitions in instances. For example, the marry(this,?partner) transitions respond to the occurrence of a marry event with concrete parameter values. Only the life cycles with this equal to the first parameter make a transition and store the value of the second parameter in the partner variable.
The transition birth(this) is also a normal transition, responding to the occurrence of a birth event. We imagine an infinite number of new Person instances to be ready in a 'prenatal' state, and only the one with the right identity stored in the special variable this makes the transition to the Single state; this is also known as instance creation.
Formally, the order of parameters is important, but for brevity's sake we shall assume the marry event to be an exception. Then, if a marry event with concrete parameters occurs, like marry(Tarzan,Jane), it is observed by both the life cycle instances Tarzan and Jane that respond by each making a transition, either from Single to Married or from Working to OverWorked.
If one life cycle can observe the occurrence of events, then so can two or more. For instance, the life cycle in Fig. 2 observes an event hire that is also observed by persons; if two life cycles respond to the same event, they are said to *communicate* about hiring personnel.
This company life cycle sets strict procedures regarding hiring of new personnel, in an attempt to avoid that partners can form an information leak to competitors.

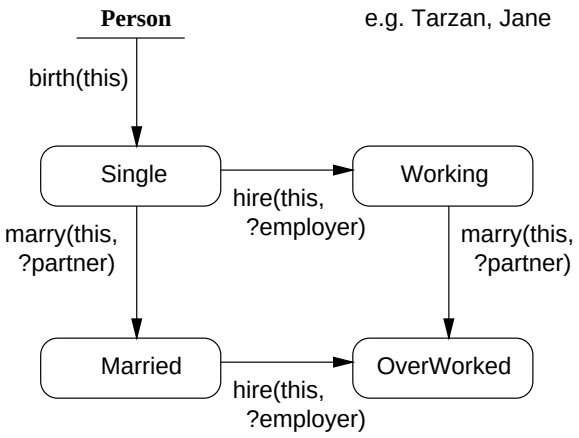


**Fig. 1.** Life cycle for a Person, who can marry and be hired.

So, Single people can be hired, or married people with jobless (that is, Married instead of OverWorked) partners, or people whose partner already works for the hiring company. These constraints are captured in the constraints in square brackets; a transition can only take place if that condition is not violated.
We are rounding up the definition of the formal semantics of life cycles at this time. Having such a semantics is a big advantage with respect to state charts, and so is the explicit creational construct, and the simplicity of the notation, which nevertheless is quite powerful.
In comparison to process algebras such as ACP [BW90], we see the advantage of a pleasing graphical notation, and a set of practical extensions such as parameters and variables and, of course, a straightforward creational construct. In comparison to the $\pi$-calculus, this paper demonstrates that our richer notation will be beneficial in proofs and providing useful feedback. We also dislike the point-to-point communication structures in the $\pi$-calculus, and prefer a CSP-like scheme [Hoa85] of globally visible events, because this has a less 'procedural' feeling.

## 2 On Process Verification

It is our intention to perform verification on process structures such as our life cycles. In our example, it would be interesting to find out if the hiring procedures practiced by companies ensure that no married couple can ever have partners working for different companies. In more formal notation, we would require that

$$\forall p, p' : \mathsf{Person} \cdot p.\mathsf{partner} = p' \\ \Rightarrow p@\mathsf{Married} \lor p'@\mathsf{Married} \lor p.\mathsf{employer} = p'.\mathsf{employer}$$

where $p@s$ indicates that life cycle instance $p$ is in state $s$.

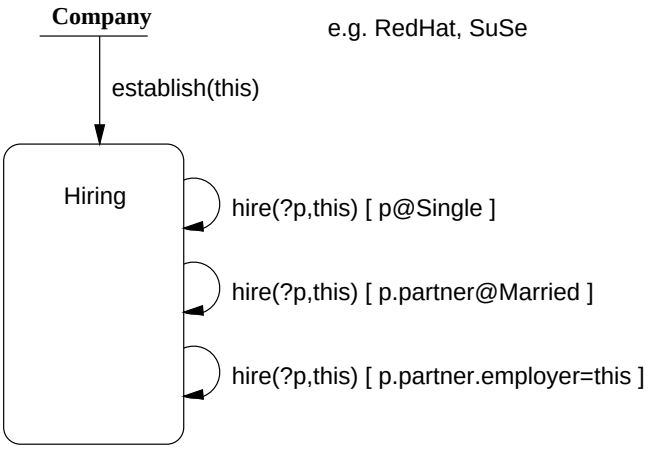


**Fig. 2.** Life cycle of a Company that fights spying.

*Feedback as a proof goal.* Such verifications are preferably done with model checkers, because they can provide excellent feedback. In this case, there is one way of invalidating the constraint, and a good model checker could produce output as in Fig. 3. This trace shows what can go wrong: If two persons are first hired by different companies and then marry, the constraint is invalidated. Such feedback is quite useful to improve designs or constraints, or perhaps make assumptions explicit, until they are accepted by the verifier.

A disadvantage of model checkers is their inability to deal with an arbitrary number of instances, as suggested by the creational construct for life cycle instances and by the $\forall p, p' : \mathsf{Person}$ quantifications. To deal with this more general class of problems, it is necessary to use more general problem solvers, and theorem provers are the customary next guess.

Theorem provers may be able to find errors like these, but they mostly provide rather crude feedback, sometimes no more than 'nope'. As we showed before [Rei99], useful feedback is crucial, and therefore we cannot accept such crude feedback. Some theorem provers will explain the paths of reasoning that led to the failure to prove a property, but these paths are usually formulated in terms of the logic used internally. This kind of feedback is too detailed to be of practical use.

*Using theorem provers.* The benefit of model checkers is that they perform their proofs at the level at which the designer thinks, namely in terms of process instances, states and transitions. When a theorem prover is taught to reason at this level, it should not be hard to provide feedback of the same quality as model checkers do, but for a more general class of constraints, including dynamic creation of new process instances.

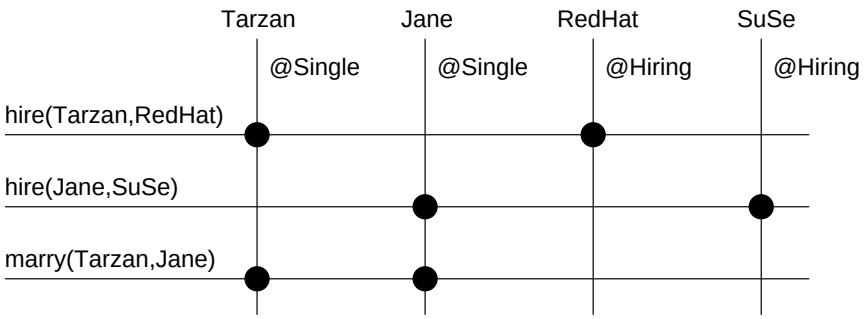


**Fig. 3.** Possible feedback from a good constraint verifier.

We want to remark explicitly that this benefit of good feedback can hardly be expected from a proof on $\pi$-calculus formulations. Although $\pi$-calculus is an excellent vehicle to specify a minimalistic semantics, it is less suitable for feedback-generation, because its syntax is too terse. For example, it is a non-trivial problem to distinguish an object from a message in a $\pi$-calculus system if both are described as a process. This has been another motivation for our self-made process language: We wanted to provide a sufficiently expressive language in which we could recognise the concepts in terms of which a process designer thinks.

To let a theorem prover 'reason' in these terms, there must be a (preferably complete, but at least sound) set of rules that describe knowledge and reasoning at that level. Except for standard logical manipulations, these rules should describe transitions, and the knowledge that can be inferred from them.

Note that this is an uncommon approach. When a syntax contains some operator $\rightarrow$ and a slightly altered variant $\rightarrow^*$, then the usual approach in theory design is to try and describe the difference between these two operators in an ad hoc operator (say, $^*$) to avoid overlapping rules. The ideal is then to make the rules entirely orthogonal and minimalistic. In our approach however, we aim for rules describing the behaviour of $\rightarrow$ as well as $\rightarrow^*$ for the simple reason that those cover the concepts in the designer's mind better.

For example, model checkers usually treat communication $\parallel$ as syntactical sugar that can be simplified with rules like this one from ACP:

$$X \parallel Y = X \between Y + Y \between X + X \mid Y$$

More rules follow to simplify the forms $\mid$ and $\between$, until nothing remains but $+$ and $.$ operators. This is useful from the perspective of definition of semantics, but it may be harmful for feedback from proofs. For example, the left merge operator $\between$ starts with an action from the left side argument, so that $\between$ does not commute, while $\parallel$ does commute: That information is lost. Another disadvantage is that the *state space* that describes the processes explode in size; this is the well-known problem of state space explosion in model checking. Retaining the $\parallel$ operator

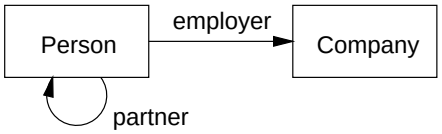


**Fig. 4.** Life cycles pointing to each other by way of variables.

can solve this problem, and that is roughly what partiality techniques [God96] in model checkers do. Getting rid of the $\|$ operator also makes it impossible to deal with process instantiation, because a state space describing any number of instantiations must be infinite-sized. The translation of derived operators to 'base' operators is useful to establish the derived operator's semantics, so that added syntax does not lead to ambiguity. In conclusion, we prefer a richer syntax and non-orthogonal proofs over terse rules applied to a terse syntax.

*Termination of proofs.* A problem of theorem provers is that they risk ending up in infinitely recursive proofs, by instantiation a quantor introduction rule over and over again. When the level of reasoning is that of logic, it cannot be foreseen how many instantiations of such introduction rules are needed to complete a proof. We expect that the added knowledge of life cycle semantics makes it possible to constrain the number of subsequent introductions and still cover most practical applications.
As explained before, a life cycle instance refers to other life cycle instances. This means [Rei00] [Kri94] that something like an entity-relationship diagram can be inferred from a life cycle definition. For example, the process references for the example of this paper are given in Fig. 4.
Our proposal is to constrain the class of solvable problems for now:

> *A theorem prover only needs to find proofs for which it traverses every relation between life cycles at most once.*

Since the number of arrows from and to a life cycle $L$ is known, it is possible to set a maximum to the number of instantiations of every quantor introduction rule of the form $\forall l : L \cdot \ldots$ and therefore, the number of instances reasoned about is also constrained, like in model checking, but the difference is that those instances can now be treated as general cases that describe the behaviour of every single instance.
We think that most practical applications lie within the boundaries endowed by the constraint on solvable problems. Our use of parameters to match event occurrences allows a set of life cycle instances to respond simultaneously, thus taking care of most set updates. Among the unsupported set updates are those for which the sets require a transitive closure, like when an employee wishes to address all managers bossing him around. These problems also occur in SQL, where they are only rarely missed. Since we know of no other limitations, we

expect that our boundaries on provable problems allow verifications on a useful class of problems.

## 3 Towards a new Behavioural Language

Model checkers usually use a language called CTL [Eme90], or the slightly more general $\mu$-calculus, to represent behavioural expressions. CTL allows quantifications over different paths to the future *and* over different moments on such paths. Several variations on CTL exist; we specifically like ACTL [DNFGR93], because it adds knowledge of the actions that cause a state transition, while CTL only considers the from and to state of a transition. As explained before, we prefer an rich language over a minimal one, to enhance the producible feedback; we even allow multi-actions [Bla96] (which we shall call multi-events) on each transition. Something that we need to add to a model checking language is quantification over instances of a life cycle $L$. For the rest, we shall try to be compatible with CTL or ACTL. We are currently defining a Behavioural Inference Logic for Life cycles, or Bill for short, that captures all such concepts.
Let $s$ be a state name. Let $v$ be a variable name (bound by a quantor), then a parameter list $p$ is a comma-separated list of variable names. An event can be written as $e(p)$, and a multi-event $m$ as a set of |-separated events. Then an expression in Bill takes the following form:

$$\begin{aligned} B ::= {} & v = v' \\ | \ & v@s \\ | \ & v^\dagger \\ | \ & \neg v \\ | \ & B \wedge B' \\ | \ & [m]\ B \\ | \ & \forall v : L \cdot B \end{aligned}$$

And the usual derived forms $\vee$, $\exists$, $\Rightarrow$, and $\neq$. The form $v@s$ is a test whether $v$ is (at the considered moment) in state $s$. The form $v^\dagger$ is a test whether $v$ is not an existing instance (it is either in the start state or in the end state). The form $[m]\ B$ is a test that $B$ holds after all possible occurrences of multi-event $m$. The remaining forms are defined as usual.
A transition like the birth of a Person can be modelled in Bill as follows:

$$\forall p : \mathsf{Person} \cdot p^\dagger \Rightarrow [\mathsf{birth}(p)]\ p@\mathsf{Single}$$

In words, when a Person, say $p$, does not exist, then the event birth($p$) will create it in the Single state.
The semantics of each life cycle $L$ can be formulated in a Bill expression $C_L$. There may also be assumptions $A$ and expectations $E$ for a system composed of life cycles. The task of the theorem prover is to prove that

$$A \ \wedge \ \bigwedge_L C_L \Rightarrow E$$

If this does not hold, good feedback must be provided. Errors can be repaired by altering any of the components of the above proposition: An error can be repaired by altering the life cycle diagrams, or by lowering expectations, or by making assumptions explicit.
Our current work is to set up a theory to reason at the level of Bill and to prove it sound and (for the selected class of problems) complete; this can then be used to tell a theorem prover how to reason at the level of Bill.

*Example 1.* Given the transitions on a Person, or $C_{\mathsf{Person}}$ in the formal model, it should be possible to derive a Bill expression $E$ like:

$$\forall p : \mathsf{Person} \cdot \neg p^{\dagger} \Rightarrow \text{in some future state, } (p@\mathsf{Married} \vee p@\mathsf{OverWorked})$$

where the form 'in some future state' is a CTL styled operator that we shall not get into here.

*Example 2.* Given the birth transition above, and given the absence of a birth transition elsewhere in the Person life cycle, the following constraint specifies that identities are unique:

$$[\ \mathsf{birth}(p)\ ]\ [\ \mathsf{birth}(q)\ ]\ p \neq q$$

Two distinctly birth event occurrences for $p$ and $q$ imply that they must represent distinct instances. Intuitively, this is quite clear, and formally it is easy to see too. However, describing such knowledge at a purely logical level is very complicated, because it is much harder to find and to formulate this particular case. The generality of the logical level comes at the price of harder specification; at the level of Bill, it is often more straightforward to see a truth.

## 4 Related Work

A lot of excellent research has been performed in the field of model checking [McM93] [BCM+90], but it is focussed on hardware verification. This means that is optimised for processes with fixed numbers of instances, quite unlike software processes.
There has been interesting work on verification of software with theorem provers [Spe99], but it suffers from feedback which is unusable in practice.

## 5 Conclusion

This paper introduced our planned approach to process verification based on our process language called *life cycles*. It is our intention to support verification on life cycles in such a way that useful feedback can be generated, and we decided to use a theorem prover to do this.
Our preliminary conclusions are that it is important to conduct such proofs on the level at which a designer thinks and specifies. This means that the proof tool

represents intermediate proofs in a syntax to which the designer can relate. It also means that the proof rules are proof steps that a designer would have made manually.
This relatively high level contains information to guide the proof strategy through decisions that would not be possible at a lower, logical level. Furthermore, a high level proof makes it possible to generate more useful feedback to a designer. However, to allow useful feedback, we have to prefer a rich syntax over a minimal one, and express rules with semantical overlap. The disadvantage of such a set of proof rules is that it cannot be designed to be terse and orthogonal.
We introduced a class of problems that we intend to address; we believe this class to be of practical use, and we explained why we believe that theorem provers can complete all proofs for this class of problems in finite time.
We introduced a behavioural language named Bill, which will hopefully permit us to let a theorem prover reason at the level of life cycle behaviour.

*Acknowledgement.* I wish to thank Maarten Fokkinga for his constant feedback and guidance of my work.

## References

[BCM+90] J.R. Burch, E.M. Clarke, K.K. McMillian, D.L. Dill, and J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In *LICS*, 1990.

[Bla96] J.O. Blanco. *The State Operator in Process Algebra.* PhD thesis, Eindhoven University of Technology, 1996.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[DNFGR93] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. In *Computer Networks and ISDN Systems*, volume 25, pages 761–778. North-Holland, 1993.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems — An Approach to the State-Explosion Problem.* LNCS 1032. Springer, 1996.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Kri94] G. Kristen. *Object Orientation: The KISS Method: From Information Architecture to Information System.* Addison-Wesley, 1994.

[McM93] K.L. McMillian. *Symbolic Model Checking.* Kluwer Academic, 1993.

[Rei99] R. van Rein. Protocol-safe workflow support for Santa Claus. In A. Vallecio, J. Hernández, and J.M. Troya, editors, *ECOOP'99 Workshop on Object Interoperability.* Universidad de Málaga, Dpto. Lenguajes y C. de la Computación, 1999.

[Rei00] R. van Rein. Specifying processes with dynamic life cycles. In *Proceedings of the CAiSE*00 conference*, pages 190–208, 2000.

[Spe99] D. Spelt. *Verification Support for Object Database Design.* PhD thesis, University of Twente, Sep 1999.