REFINER: Towards Formal Verification of Model Transformations

Anton Wijs and Luc Engelen

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{A.J.Wijs,L.J.P.Engelen}@tue.nl

Abstract. We present the REFINER tool, which offers techniques to define behavioural transformations applicable on formal models of concurrent systems, reason about semantics preservation and the preservation of safety and liveness properties of such transformations, and apply them on models. Behavioural transformations allow to change the potential behaviour of systems. This is useful for model-driven development approaches, where systems are designed and created by first developing an abstract model, and iteratively refining this model until it is concrete enough to automatically generate source code from it. Properties that hold on the initial model and should remain valid throughout the development in later models can be maintained, by which the effort of verifying those properties over and over again is avoided. The tool integrates with the existing model checking toolsets MCRL2 and CADP, resulting in a complete model checking approach for model-driven system development.

1 Introduction

REFINER[1] is a tool to verify so-called *behavioural* transformations of formal models of concurrent systems. Such transformations allow to manipulate the potential behaviour of the processes in a model. The ability to verify them opens up the possibility to step-wise develop complex concurrent systems, while preserving important system properties. Step-wise system development allows a developer to start the design phase with an abstract model, and making it more and more concrete through small, manageable transformations, until a model has been obtained with sufficient information to generate source code from it.

With REFINER, a developer can construct behavioural transformations, which the tool can efficiently analyse to determine if it preserves the semantics of models it is applied on, and if it preserves given safety or liveness properties. To the best of our knowledge, this is the first tool that can automatically check property preservation of user-defined model transformations, independent of source models. The topic is related to refinement checking. However, tools such as RODIN [1], FDR2,[2]

[1] Available at http://www.win.tue.nl/~awijs/refiner
[2] http://www.fsel.com/documentation/fdr2/html

CSP-CASL-PROVER [2] can establish refinements between given models, but not verify transformation rules. ATELIER B[3] uses a notion comparable to transformation rules, but verifies resulting models instead of the rules themselves.

Semantics and property preservation checking is done by a single analysis technique. The first case is useful for refactoring and restructuring of models, while the second one allows for behaviour refinements. The technique is independent of the input and output models; it does not involve the state space of either of them, hence it usually works many orders of magnitude faster than repeated verification of the models through standard model checking, and it allows to build a repository of verified transformations. The tool integrates with the action-based, explicit-state model checking toolsets CADP [3] and MCRL2 [4]. These tools can be used to model concurrent systems in process algebras and automata, and to verify that the models satisfy functional properties. The semantics of the processes in such models can be represented by *Labelled Transition Systems* (LTSs), and the process LTSs can be combined using synchronisation composition. In REFINER, transformations are formalised as LTS transformations, defining which patterns in the LTSs need to be transformed into particular new patterns.

The theoretical basis has been published as [5–7]. Since then, a prototype implementation has been further developed to a complete tool, with a graphical user interface and multi-core computation capability.

2 Models and Model Transformations

REFINER uses a compositional action-based formalisation of system behaviour, i.e. LTSs are used to define the potential behaviour of individual processes and of systems as a whole. Its techniques are therefore applicable on any model with an LTS semantics, e.g. expressed in a process algebra. An LTS is a quadruple $\mathcal{G} = \langle \mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \underline{s}_\mathcal{G} \rangle$, with $\underline{s}_\mathcal{G}$ the initial state, $\mathcal{S}_\mathcal{G}$ the (finite) set of states reachable from $\underline{s}_\mathcal{G}$, $\mathcal{A}_\mathcal{G}$ a set of actions used to identify events, $\tau \notin \mathcal{A}_\mathcal{G}$ being a special action representing internal events, and $\mathcal{T}_\mathcal{G} : \mathcal{S}_\mathcal{G} \times \mathcal{A}_\mathcal{G} \cup \{\tau\} \times \mathcal{S}_\mathcal{G}$ a relation expressing which actions can be performed in which states, and what the resulting state is. With $s \xrightarrow{a}_\mathcal{G} s'$, we express that $\langle s, a, s' \rangle \in \mathcal{T}_\mathcal{G}$.

Process LTSs can be combined into a system. This is formalised as a *network of* LTSs [8]. In the following, given an integer $n > 0$, $1..n$ is the set of integers ranging from 1 to n. A vector $\overline{v}$ of size n contains n elements indexed by $1..n$. For $i \in 1..n$, $\overline{v}[i]$ denotes element i in $\overline{v}$.

Definition 1 (Network of LTSs). *A* network of LTSs $\mathcal{M}$ *of size* n *is a pair* $\langle \mathit{\Pi}, \mathcal{V} \rangle$*, where*

- $\mathit{\Pi}$ *is a vector of* n *(process) LTSs. For each* $i \in 1..n$*, we write* $\mathit{\Pi}[i] = \langle \mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, \mathcal{I}_i \rangle$*, and* $s_1 \xrightarrow{b}_i s_2$ *is shorthand for* $s_1 \xrightarrow{b}_{\mathit{\Pi}[i]} s_2$*;*
- $\mathcal{V}$ *is a finite set of synchronisation laws. A synchronisation law is a tuple* $\langle \overline{t}, a \rangle$*, where* a *is an action label, and* $\overline{t}$ *is a vector of size* n *called a* synchronisation vector*, in which for all* $i \in 1..n$*,* $\overline{t}[i] \in \mathcal{A}_i \cup \{\bullet\}$*, where* $\bullet$ *is a special symbol denoting that* $\mathit{\Pi}[i]$ *performs no action.*

[3] `http://www.atelierb.eu`

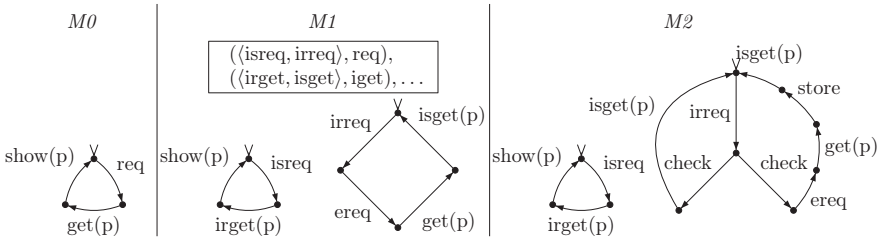

Fig. 1. Three versions of a network modelling an agent fetching pages

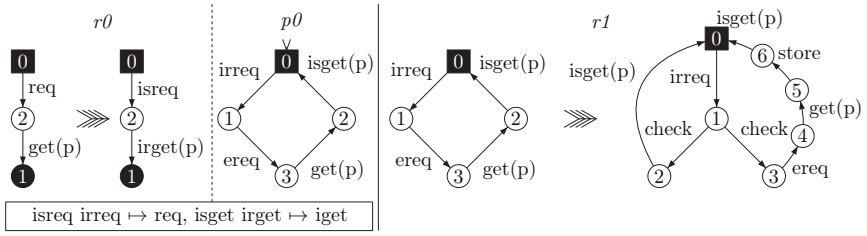

Fig. 2. Rules r_0, p_0 of rule system R_0, and r_1 of rule system R_1

The synchronous composition LTS($\mathcal{M}$) defined by a network $\mathcal{M}$ represents the state space of $\mathcal{M}$, and is an LTS with $\underline{s} = \langle \underline{s}_1, \ldots, \underline{s}_n \rangle$, $\mathcal{A} = \{a \mid \langle \overline{t}, a \rangle \in \mathcal{V}\}$, $\mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$, and $\mathcal{T}$ is the smallest relation satisfying:

$$\langle \overline{t}, a \rangle \in \mathcal{V} \wedge (\forall i \in 1..n) \left(\begin{array}{c} (\overline{t}[i] = \bullet \wedge \overline{s}'[i] = \overline{s}[i]) \\ \vee\ (\overline{t}[i] \neq \bullet \wedge \overline{s}[i] \xrightarrow{\overline{t}[i]}_i \overline{s}'[i]) \end{array} \right) \implies \overline{s} \xrightarrow{a} \overline{s}'.$$

We formalise behavioural model-to-model transformations from networks of LTSs to new networks of LTSs as *rule systems*, containing a finite number of LTS *transformation rules*. Such a rule consists of a pair of LTSs $\mathcal{L} \rightarrow \mathcal{R}$. The used transformation mechanism is the *double-pushout* method from graph transformation [9]: $\mathcal{L}$ defines a pattern, to be found and replaced in a given LTS $\mathcal{G}$ (for this, a *match*, i.e. an injective homomorphism, $m : \mathcal{L} \rightarrow \mathcal{G}$ must be established), and $\mathcal{R}$ defines the pattern that should replace all occurrences of $\mathcal{L}$ in $\mathcal{G}$. Apart from some conditions that need to hold in order to have a valid match of $\mathcal{L}$ on an LTS $\mathcal{G}$,[4] a subset of so-called *glue-states* $S \subseteq \mathcal{S}_\mathcal{L} \cap \mathcal{S}_\mathcal{R}$ is defined, which indicates how $\mathcal{L}$ relates to $\mathcal{R}$. When applying transformation on a match $m : \mathcal{L} \rightarrow \mathcal{G}$, resulting in a new LTS $T(\mathcal{G})$, first, all states matched on $\mathcal{S}_\mathcal{L} \setminus S$ and all related transitions are removed, and second, each state in $\mathcal{S}_\mathcal{R} \setminus S$ (and related transitions) leads to a new state in $\mathcal{S}_{T(\mathcal{G})}$ (and new related transitions in $\mathcal{T}_{T(\mathcal{G})}$).

Rules are applied on process LTSs of a network to transform it, but rule systems also include left and right synchronisation laws, expressing how behaviour in the left and right rule patterns, respectively, should synchronise with each other and the outside world. In order for a rule system R to be applicable on a network $\mathcal{M}$, the left laws of R must be compatible with those of $\mathcal{M}$, and if so, then the right laws of R are introduced when transforming.

[4] The interested reader is referred to [6, 9].

Figs. 1 and 2 show a small, but illustrative example of the approach. In Fig. 1, network M_0 is an abstract specification of an agent, for instance a web browser, which can request a page (*req*), receive a page p (*get(p)*), and display it (*show(p)*). Initial states of LTSs are marked with an incoming arrowhead. Actually, M_0 is still very abstract, and we wish to specify that the communication with the outside world is handled by an additional component. This is added in M_1, and we have two new laws expressing the need for synchronisation between the two components over actions internal to the system (these actions are prefixed by 'i'). We can obtain M_1 from M_0 by transforming the latter using a rule system R_0 defined in Fig. 2. There, black states are glue-states, and square black states are glue-states with the added condition that states matched on them do not have outgoing transitions that are not covered by the left pattern. Rule r_0 rewrites the component we already had in M_0, and p_0 is a special kind of rule called a *process adding* rule, which adds a new component. It can be interpreted as a rule with an empty left pattern. Finally, it introduces two new laws, expressed without using vectors, since the rules have no fixed order. When transforming, these are matched on the input network to derive concrete new laws for the new network.

Likewise, M_1 can be transformed to M_2 with the motivation that the communication component should have a local buffer, and check for each request whether that page is already in the buffer before attempting communication with the outside world. Rule r_1 of Fig. 2 can be applied on M_1 to obtain M_2.

In this example, the networks are not much larger than the rule systems, but in practice, they usually are, and rules are often applicable in multiple places.

Verification of Transformations. REFINER can check whether a rule system R is confluent, i.e. leads to a unique target model, and verify whether it is semantics preserving and/or correctness preserving, i.e. that it preserves a desired system property. In both cases, it identifies, based on the left and right laws of R, which transformation rules are dependent on each other. Two rules $r = \mathcal{L}_r \rightarrow \mathcal{R}_r$, $r' = \mathcal{L}_{r'} \rightarrow \mathcal{R}_{r'}$ are dependent iff in $\mathcal{L}_r$ (or $\mathcal{R}_r$), there is at least one transition that needs to synchronise with a transition in $\mathcal{L}_{r'}$ (or $\mathcal{R}_{r'}$). This partitions the set of rules in R into sets of dependent rules. For each set D of dependent rules,[5] the left patterns and the right patterns of all the elements are combined into two new networks D_L and D_R. For semantics preservation, it is checked if LTS(D_L) and LTS(D_R) are *(strongly) bisimilar*, i.e. whether they can be considered equivalent.

A more general approach is required to check the preservation of particular properties. In order to allow the semantics to be altered, D_L and D_R should be compared w.r.t. a given property, instead of the entire semantics. For this, we move the LTS(D_L) and LTS(D_R) to an appropriate level of abstraction before the analysis, using the *maximal hiding* technique [10]. For any property φ written in the μ-calculus fragment L_μ^{dsbr} [10], maximal hiding hides all actions in an LTS, i.e. renames them to τ, that are not crucial for the truth-value of φ. Furthermore, it is shown in [10] that if φ is satisfied by an LTS $\mathcal{G}_1$, and $\mathcal{G}_1$ is *divergence-sensitive*

[5] In addition to each D, also all their subsets are involved in the analysis, the latter representing situations with unsuccessful synchronisation. For the details, see [6].

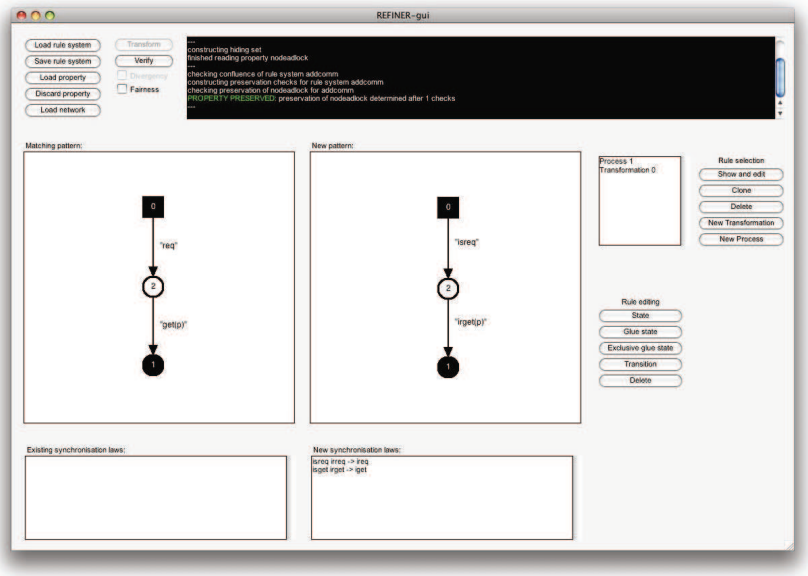

Fig. 3. The graphical user interface of REFINER

branching bisimilar [11] (DSBB)[6] to an LTS $\mathcal{G}_2$, then also $\mathcal{G}_2$ satisfies φ. This allows comparing LTSs w.r.t. φ. We apply maximal hiding w.r.t. a given φ to the LTS(D_L) and LTS(D_R), before checking that they are DSBB. If the checks pass for all D_L and D_R, then R preserves φ. Semantics preservation checking is actually the special case in which maximal hiding has no effect.

Consider again the example rule systems of Fig. 2, and the L_μ^{dsbr} property $\varphi = [\mathsf{true}^*][\tau^*.req]([(\neg get(p))^*]\neg\text{deadlock} \wedge [\neg get(p)]\ \dashv)$, where 'deadlock' is a formula expressing the presence of a deadlock. This property expresses that after every *req*, eventually a *get(p)* will occur (for the semantics of L_μ^{dsbr}, see [10]). Based on this, maximal hiding will hide all transition labels, except for *req* and *get(p)*. Combining the left- and right-patterns of r_0 and p_0, constructing the synchronous compositions, and applying this hiding, leads to DSBB LTSs. This is also the case for r_1 in isolation. Hence, both rule systems preserve φ.

3 Implementation

REFINER has been implemented in PYTHON 3, and consists of about 5,000 lines of code. It is platform-independent, and has a graphical user interface (Fig. 3), implemented using the TKINTER module, but it can also be run from the command line. It provides the functionality to create and edit rule systems, load and save them, apply them on models, and verify them in various ways. The tool does not focus on creating and verifying models; instead, this can be done using the model checker toolsets CADP and MCRL2. With these tools, REFINER shares file formats for LTSs, networks of LTSs, and L_μ^{dsbr} properties.

For verification, REFINER allows to check semantics preservation of rule systems, model-independent preservation of properties, and property preservation

[6] DSBB is, like weak and branching bisimilarity [11], sensitive to τ-transitions, but also to divergences, i.e. the ability to perform infinite sequences of τ-transitions [11]. As such, it preserves safety and liveness properties.

w.r.t. particular models. Besides this, there are the fine-tuning options 'fairness', by which divergences in a rule system will be ignored (useful for safety properties), and 'divergency', by which the divergences already present in an input network will be taken into account, allowing for a more relaxed check.

Finally, Refiner has multi-core computation capability. Verifying a rule system may involve many DSBB checks. Since these can be done independently, parallelisation is straight-forward. Experiments have shown that this scales linearly [7]. As demonstrated in [6], Refiner can, through model transformation verification, determine in mere seconds that transformed networks with state spaces of multiple billions of states satisfy a particular property. Model checking those networks would take many orders of magnitude more time. For future work, we plan to support timing [12], and directed search techniques [13].

References

1. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. STTT 12(6), 447–466 (2010)
2. Kahsai, T., Roggenbach, M.: Property Preserving Refinement for Csp-Casl. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 206–220. Springer, Heidelberg (2009)
3. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
4. Cranen, S., Groote, J., Keiren, J., Stappers, F., de Vink, E., Wesselink, W., Willemse, T.: An Overview of the mCRL2 Toolset and Its Recent Advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013)
5. Engelen, L., Wijs, A.: Incremental Formal Verification for Model Refining. In: MoDeVVa 2012, pp. 29–34. ACM (2012)
6. Wijs, A., Engelen, L.: Efficient Property Preservation Checking of Model Refinements. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 565–579. Springer, Heidelberg (2013)
7. Wijs, A.: Define, Verify, Refine: Correct Composition and Transformation of Concurrent System Semantics. In: Xue, J., Fiadeiro, J.L., Liu, Z. (eds.) FACS 2013. LNCS, Springer (2013) (to appear)
8. Lang, F.: Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-Fly Verification Methods. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)
9. Heckel, R.: Graph Transformation in a Nutshell. In: FoVMT 2004. ENTCS, vol. 148, pp. 187–198. Elsevier (2006)
10. Mateescu, R., Wijs, A.: Property-Dependent Reductions for the Modal Mu-Calculus. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 2–19. Springer, Heidelberg (2011)
11. van Glabbeek, R., Weijland, W.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM 43(3), 555–600 (1996)
12. Fokkink, W., Pang, J., Wijs, A.: Is Timed Branching Bisimilarity an Equivalence Indeed? In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 258–272. Springer, Heidelberg (2005)
13. Wijs, A.: What To Do Next?: Analysing and Optimising System Behaviour in Time. PhD thesis, VU University Amsterdam (2007)