Incremental Formal Verification for Model Refining

Anton Wijs
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
A.J.Wijs@tue.nl

Luc Engelen
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
L.J.P.Engelen@tue.nl

ABSTRACT

When developing complex software systems, it is vital to ensure that the final product satisfies all the stated requirements. Model checking can help to exhaustively check models of such systems, but due to its high computation demands, it is often not practical. In this paper, we present a new technique to check that properties are preserved when a model at a high level of abstraction is refined to one at a lower level through transformations. In this way, correctness of the resulting models can be determined efficiently. This technique has been implemented, and we demonstrate its usefulness in practice.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/ Program Verification — Model checking; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages — Process models

General Terms

Verification, Reliability, Measurement

Keywords

formal verification, model refinement, property preservation

1. INTRODUCTION

When developing complex software systems, it is vital to ensure that the final product is 'correct', i.e. that it satisfies all the stated requirements. If an error is detected in an early stage of development, then it can be fixed relatively easy, while fixing errors in the final product is much more difficult and costly. To verify that a system model is indeed correct, *model checking* [5] can play an instrumental role. Unfortunately, however, its high computation demands make it often impractical. This is already the case for models consisting of a few hundred lines of description, therefore completely checking the models of real-life systems is currently next to impossible.

In practice, systems are often developed in a 'top-down' manner, meaning that initially, a model is designed at a high level of abstraction, and in several development iterations, this model gets more and more detailed. We call this *model refinement* (MR). The process of MR can be automated by expressing the refinements as model transformations. In this setting, the developer creates the initial model and defines the transformations leading to subsequent models. For instance, the initial model may explain that a system consists of several independent processes exchanging some data, a second version of the model may describe that the data must be sent in batches of ten messages, and a third model may add that the channels are lossy.

The main purpose of Model-Driven Software Development (MDSD) is to automatically derive software code from models. MR fits into this if we interpret the initial model as being at a high level of abstraction, and the end product, i.e. the code, as a model at the lowest necessary level of abstraction. When performing MDSD with MR, the code is to be obtained by iteratively refining the initial model.

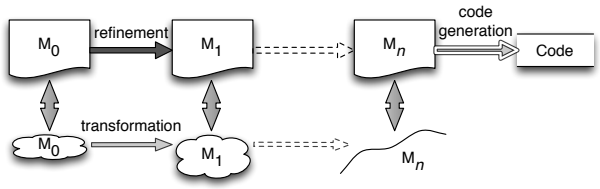

Figure 1: Iterative model refinement in relation to transformations of Labelled Transition Systems

As models gain details, it becomes increasingly hard to completely verify them. In this paper, we propose a verification technique based on *explicit-state* model checking [5], supporting the MDSD by MR software development method. It exploits the fact that a model M_{i+1}, with M_0 the initial model, and $i \in \mathbb{N}$, can be described relative to M_i, i.e. by describing the changes applied to M_i. Figure 1 presents the MDSD setting we consider in this paper. In the next section, we explain the involved notions in more detail. A model is denoted by a square with a curly bottom, and its semantics is expressed as a *Labelled Transition System* (LTS), denoted by a cloud. An LTS is a directed graph describing all potential behaviour of the modelled system. Likewise, the *refinement* of a model corresponds with the *transformation* of its LTS, so a sequence of refinement steps can be expressed as a se-

MoDeVVa '12, September 30 2012, Innsbruck, Austria

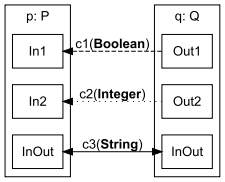

Figure 2: Communication diagram of an SLCO model

quence of transformations on the LTS level. Our verification technique actually checks that the transformation of an LTS does not interfere with the truth-value of the properties, i.e. requirements, satisfied by the LTS M_i (and therefore, also by the model M_i). If this is the case, then it follows that these properties hold for M_{i+1} as well, and complete verification of the new model can be avoided. As each LTS in a transformation sequence is likely to be much larger than its predecessor, due to the lower level of abstraction (which is depicted in Figure 1), avoiding complete rechecking is crucial for the success of our technique. Properties can be verified in an early stage of development, and their validity in subsequent models is ensured by the transformations.

In the next section, we describe a modelling language compatible with our technique, called SLCO [2], and discuss how its semantics can be expressed by *networks of* LTSs. We stress, however, that any language whose semantics can be expressed with such networks is in principle supported. Section 3 contains an explanation of our verification technique, and Section 4 presents experimental results. Finally, we discuss related work in Section 5, and Section 6 contains conclusions and pointers to future work.

2. BACKGROUND

The work presented in this paper provides a basis for the automated verification of model transformations related to languages whose semantics can be described as LTSs. For illustrative purposes, we focus on one such language, the Simple Language of Communicating Objects (SLCO) [2]. In SLCO, systems consisting of concurrent, communicating objects can be described on a high level of abstraction using an intuitive graphical syntax. By applying sequences of model transformations to various target platforms, SLCO models can be simulated, verified, and executed. The objects in an SLCO model are connected by channels, over which they communicate by sending and receiving signals. They are connected to the channels via their ports. The communication diagram in Figure 2 shows two objects p and q that are connected by three channels. Both objects have three ports. Objects in SLCO are instances of classes. Figure 2 shows that p is an instance of class P, and q is an instance of class Q. SLCO offers three types of channels: synchronous channels, asynchronous lossless channels, and asynchronous lossy channels. Synchronous channels are denoted as arrows with solid lines, asynchronous lossless channels as dashed arrows, and asynchronous lossy channels as dotted arrows. Channels are either unidirectional or bidirectional, and their directionality is indicated by the number of arrow heads. Furthermore, each channel is suited to transfer signals with a certain signature. The channel *c3*, e.g., can only be used to send and receive signals with a single string argument.

The behaviour of the instances of each class is specified using state machines, such as the ones shown in the behaviour diagram of Figure 3, which consist of a number of states and transitions. The three state machines on the left of Figure 3 belong to class P, and the state machine on the right belongs to class Q. Each transition has a source and a target state, and a list of statements that are executed when the transition from the source to the target state is made. A transition is enabled if the first of these statements is enabled. A transition without statements is also enabled. SLCO supports a variety of statement types. For communication between objects, there are statements for sending and receiving signals. The statement **send** $T(s)$ **to** *InOut*, for instance, sends a signal named T with a single argument s via port *InOut*. Its counterpart **receive** $T(s)$ **from** *InOut* receives a signal named T from port *InOut* and stores the value of the argument in variable s. Statements such as **receive** $P([[\textbf{false}]])$ **from** *In1* offer a form of conditional signal reception. Only those signals whose argument is equal to **false** will be accepted. There is also a more general form of conditional signal reception. For example, statement **receive** $Q(m \mid m \geq 0)$ **from** *In2* only accepts those signals whose argument is at least equal to 0. Boolean expressions, such as $m == 6$, denote statements that block until the expression holds. Time is incorporated in SLCO by means of delay statements. For example, the statement **after** 5 **ms** blocks until 5 ms have passed. Assignment statements, such as $m := m + 1$, are used to assign values to variables. Variables either belong to a class or a state machine. The global variables that belong to a class are accessible by all state machines that are part of the class, whereas local variables are only accessible by the state machine they belong to.

As mentioned above, various model transformations have been developed to transform SLCO models to other plat-

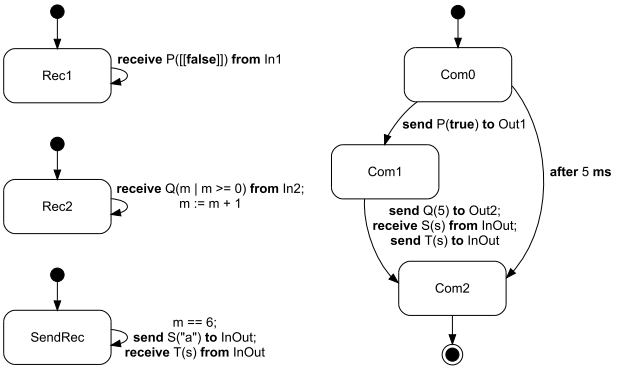

Figure 3: Behaviour diagram of an SLCO model

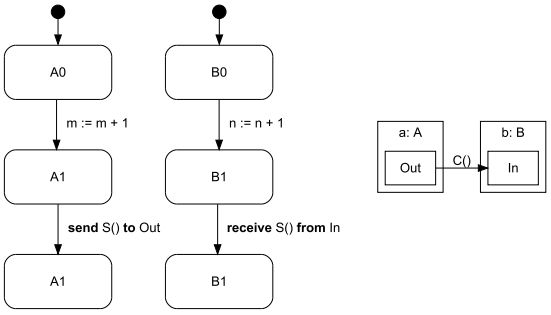

Figure 4: Communicating objects in SLCO

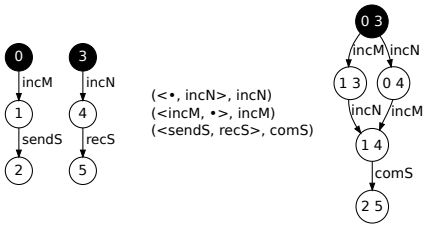

Figure 5: Communicating LTSs

forms. The large majority of these transformations are refining, endogenous transformations [2] that transform a given SLCO model to a more concrete SLCO model. The purpose of these transformations is to lower the level of abstraction of models, without altering the observable behaviour of the systems they describe. Generally, when two objects communicate with each other, modifying one without the other leads to models with undesirable behaviour. To keep the observable behaviour intact after refinement, all communicating objects must be modified. For example, the model shown in Figure 4 consists of two objects a and b that communicate over a channel named C. The leftmost state machine specifies the behaviour of a, and the rightmost state machine specifies the behaviour of b. Modifying the name of the signal sent by a without modifying b leads to a model in which communication is unsuccessful.

The semantics of an SLCO model can be expressed in an LTS, which we call the model LTS. Such an LTS consists of states and labelled transitions between states describing state changes of the model, where the labels indicate what action the system performs. Unfortunately, a linear growth of the size of a model tends to lead to an exponential growth of the model LTS. This is known as the *state space explosion problem*. However, if a model consists of several concurrent processes, it is also possible to express its behaviour as a *network of* LTSs [15], thus avoiding this problem. In such a network, the semantics of each process is represented by a process LTS, and *synchronisation rules* define how the actions of the process LTSs interact with each other. In that way, the LTSs in a network, together with the synchronisation rules, implicitly define the model LTS.

On the left of Figure 5, two LTSs are shown that represent the SLCO model of Figure 4 on the LTS level. Each transition has a source and a target state, and is labeled with an action. A subset of the states of an LTS are designated as initial states. In Figure 5, the states are depicted by numbered circles, and the transitions are represented by arrows. The initial states are coloured black. The statements of the model in Figure 4 are represented by abstract actions in Figure 5. To form the model LTS from these two process LTSs, synchronisation rules are required. These rules define which actions in the process LTSs synchronise with each other. They are shown in the middle of Figure 5, where a • indicates 'no action'. The synchronisation rule that defines that action *sendS* of the leftmost LTS synchronises with action *recS* of the rightmost LTS, leading to an action *comS* in the model LTS, is as follows: $(\langle sendS, recS\rangle, comS)$. This rule implies that the two process LTSs are related and that modifying one without the other might lead to undesired results. The model LTS obtained by combining the process LTSs according to the synchronisation rules is shown on the right of Figure 5.

For explicit-state model checking, usually a model LTS is analysed to determine whether a temporal logic formula φ, formalising a model requirement, is satisfied. As model LTSs grow exponentially when applying MR, we propose a technique to avoid model checking of refined models. Instead, it determines whether a refinement step preserves φ. Then, if φ holds for a given model, it also holds after refinement.

3. APPROACH

As already mentioned in Section 1, the refinement of a model is represented at the LTS level by a transformation. We formalise such a transformation by a set of n non-conflicting[1] LTS *transformation rules* $r_1, \ldots, r_n$ $(n \in \mathbb{N})$, together called a *rule system*. Figure 6 illustrates this concept. A rule system also includes a set of synchronisation rules, similar to the ones in a network of LTSs, describing how the actions in the rules synchronise. A rule system is applicable on an input network if there are no contradictions between the synchronisation rules of the rule system and the network.

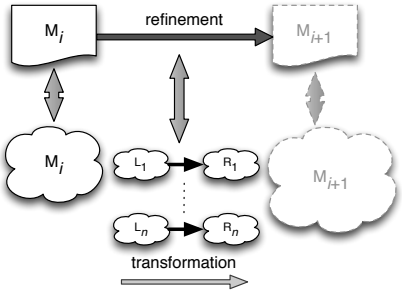

Figure 6: LTS transformation as a rule system

A transformation rule r_j (with $1 \leq j \leq n$) is a pair of LTSs $\langle L_j, R_j\rangle$. When transforming an input LTS M_i, for each r_j in the rule system, all matches of L_j in M_i, i.e. all subgraphs which are isomorphic with L_j, are replaced by the corresponding R_j. States that appear both in L_j and R_j (states are uniquely identified by an ID) are the *glue-states*, relative to which all changes are applied. For a complete formalisation of this, see [8]. It is a basic, yet expressive way to reason about refinements, allowing to express the transformation of model behaviour in general. Moreover, by representing refinements at the LTS level, we ensure that the verification technique is applicable for all modelling languages for which the semantics can be expressed with LTSs, e.g. SLCO.

Transformation is actually done before the model LTS is constructed, i.e. it is applicable on the process LTSs in a network. This is illustrated in Figure 7, where a model consisting of m processes (rounded squares) is represented by a network of LTSs $P_1, \ldots, P_m$. Transformation rules are applicable on individual processes, thus applying all the rules in a rule system on a network of LTSs yields a new network, which in term defines a new model LTS. To also allow transformation of the synchronising behaviour in the network, a rule system may introduce new synchronisation rules during a transformation. In that way, new interactions between processes can be defined, and existing ones can be refined.

[1] In [8], we discuss in detail the conditions such a set of rules should meet to ensure that the transformation is confluent, i.e. leads to a single, unique new network of LTSs.

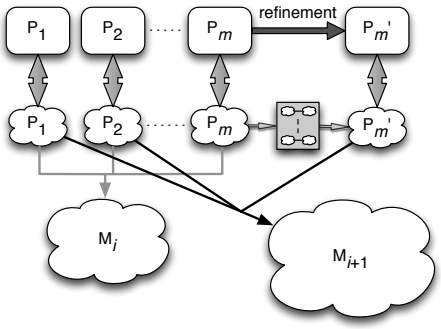

Figure 7: Refinement of an individual process affects the entire LTS network

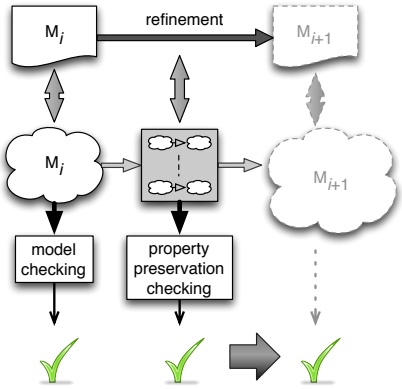

Figure 8: Deriving model checking results through property preservation of rule systems

The general approach of our verification technique is highlighted in Figure 8. For LTS transformation rule systems, we define a so-called *property preservation check*. This check computes whether the given rule system preserves a given temporal logic formula φ, *regardless* of the input network of LTSs. This enables the use of rule systems as re-usable entities; a rule system $\mathcal{R}$ preserving a formula φ can be applied on any model satisfying this formula. If a given input model M_i satisfies φ, then one can directly conclude from the fact that $\mathcal{R}$ preserves φ that φ is also satisfied by the model M_{i+1} resulting from applying $\mathcal{R}$ on M_i. A formal description of the check and a correctness proof can be found in [8].

Property preservation checking of a rule system $\mathcal{R}$ entails performing preservation checks for all complete subsets of $\mathcal{R}$ of dependent transformation rules. To illustrate this, consider the example from Figure 5, and the rule system displayed in Figure 10, where σ_L and σ_R are the sets of synchronisation rules before and after transformation, respectively. This system defines a straightforward renaming of all actions. The synchronisation rules of a rule system directly imply a dependency relation between the rules contained in it. For instance, since action *incM* can be fired independently, rule (1) is also independent from other rules, whereas rules (3) and (4) are dependent, since they involve dependent actions. Note that in general, rules can involve multiple actions, and dependencies can involve more than two rules. In this case, there are three complete sets of dependent rules, forming the partition $\Sigma = \{\{(1)\}, \{(2)\}, \{(3), (4)\}\}$. Finally,

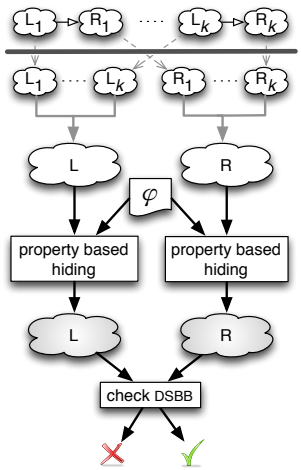

Figure 9: Property preservation checking for a set of dependent transformation rules

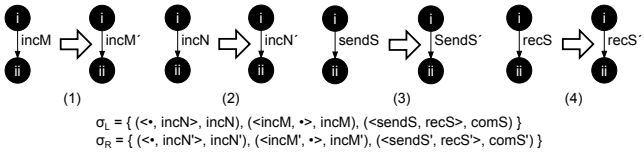

Figure 10: Transformation rules that rename actions

we add all non-empty subsets of these sets, ending up with $\Sigma' = \{\{(1)\}, \{(2)\}, \{(3)\}, \{(4)\}, \{(3), (4)\}\}$, where $\{(3)\}$ and $\{(4)\}$ represent unsuccessful synchronisation situations, i.e. where one process tries to synchronise, but the other one cannot (as is the case in states '1 3' and '0 4' in Figure 5).

Figure 9 contains the workflow of a preservation check for one element $e \in \Sigma'$. Besides e (see the top of Figure 9), the check requires a temporal logic formula φ. For $e = \{r_1, \ldots, r_k\}$, where $1 \leq k \leq n$, first, all L_j of the rules r_j $(1 \leq j \leq k)$ are combined into a new network, using the synchronisation rules in σ_L. The same is done with all R_j and σ_R. These two networks result in two LTSs L and R, representing the interaction of behaviour subjected to transformation before and after the transformation.

Next, a technique called *property based hiding* is applied on both L and R. This is a technique which, first of all, identifies the largest set of actions in the system that are irrelevant for the truth-value of φ. Stated differently, it identifies the smallest amount of system behaviour that needs to be considered in M_i to correctly determine whether φ is satisfied or not. Second of all, it renames all the irrelevant actions in the LTSs to a designated label, usually τ, effectively hiding them. In [16], a fragment of the temporal logic μ-calculus [14] was identified which is fully compatible with a flavour of bisimulation called *divergence-sensitive branching bisimulation* (DSBB) [10]. DSBB is an equivalence relation between LTSs allowing the comparison of system behaviour involving action hiding. It is, however, more discriminating than other bisimulations involving hiding, such as *branching* [10] and *weak* [18] bisimulation. Because of this, it supports a more expressive temporal logic, i.e. the fragment identified in [16]. This support, or the compatibility between the logic and DSBB, means that if a property φ

written in this logic holds in an LTS A, and A is equivalent to a second LTS B according to DSBB, then it follows that φ also holds in B. This, together with property based hiding, are crucial aspects of our technique. It is important to note that the identified fragment of the μ-calculus is expressive enough to formalise a vast majority of the properties used in practice [16].

Through property based hiding, we abstract away all behaviour in L and R irrelevant for φ (in Figure 9 represented by making their clouds grey). This maximises the potential for concluding that L and R are equivalent (according to DSBB) with respect to φ. Finally, the hidden L and R are compared using a DSBB comparison algorithm. If all the checks involved in checking whether $\mathcal{R}$ is property preserving produce a positive result, then $\mathcal{R}$ indeed preserves the property. If this is not the case, then $\mathcal{R}$ does not preserve the property for networks of LTSs *in general*, which does not exclude that there exists a particular network which can safely be transformed with $\mathcal{R}$.

4. EXPERIMENTAL RESULTS

In order to compare the performance of property preservation checking with traditional explicit-state model checking, we performed experiments on five case studies. For each case study, we defined a transformation to be applied on a given model. Table 1 shows the main results of the experiments. In the first two rows, the runtime for full verification of the initial model and the transformed model are given. The last three rows show the number of checks and the result of the property preservation check on the transformation, as well as the time it takes to perform the various bisimulation checks. Some transformations preserve properties (noted by ✓) and some do not (noted by ✗). The experiments have been performed on a machine with two dual-core AMD OPTERON (tm) processors 885 2.6 GHz, 126 GB RAM, running RED HAT 4.3.2-7. The results clearly show that checking property preservation takes much less time than verifying the transformed models. In practice, verification of a transformed model can be avoided if the transformation preserves a property. In case it does not, one can choose to define another transformation and check again, or apply the transformation and verify the transformed model. In the experiments, we validated the checks using the results of the verification of the transformed models. This was successful, i.e. property preserving transformations indeed led to transformed models satisfying the property in question.

The first three models are part of the distribution of the MCRL2 toolset [11], and their state spaces were generated with the MCRL2 tools. The other two models were created specifically for these experiments, and their state spaces were generated with EXP.OPEN [15]. Although it is possible to generate the checks for these cases using EXP.OPEN, we used a proprietary prototype tool to generate the network LTSs that form these checks. For divergence-sensitive branching bisimilarity checking of these networks, we used the *ltscompare* tool of the MCRL2 toolset.

In the *c.syst.* case, a variant of the Alternating Bit Protocol (ABP) [3] is added to a system of communicating processes in five different places. We analysed a correct and an incorrect version of the rule system that transforms models by adding this protocol. Figure 11 shows a part of the correct transformation. In this figure, action $r1a'$ denotes the reception of a signal a from channel 1. Transferring this signal over channel 2 after adding a bit set to *true* is denoted by the action *s2at*. Action *r2t* denotes the reception of the acknowledgement of the previous signal. The same encoding from communication over channels by sending and receiving signals to abstract actions is used for all actions in the figure. A detailed description of the other cases is given in [8].

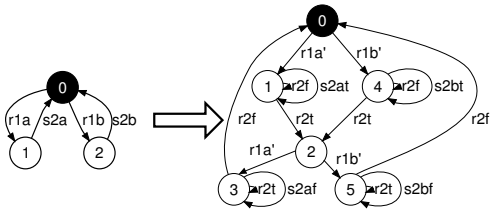

Figure 11: Transformation rule for the ABP

5. RELATED WORK

In [1, 17], classifications of model transformations and verification techniques are given. Syntactically, the transformations we consider are in-place, endogenous, and vertical, since the level of abstraction of the source model is changed. Furthermore, we focus on relations between the semantics of the source and target models. Finally, our verification technique is transformation-dependent and input-independent.

Incremental bisimilarity checking [20] is related, however we do not maintain global bisimilarity results, but derive them via the analysis of transformations. In [6, 12, 19], semantics preservation of model transformations is checked using either strong or weak bisimilarity. They, however, consider exogenous, horizontal transformation. Like [12], but unlike [6, 19], we do not involve the source and target LTSs in the check, thereby determining whether a transformation is property preserving in general.

A technique to check preservation of safety properties by horizontal transformations is presented in [24]. In contrast, we consider vertical transformations and our technique also supports liveness properties, which are harder to check.

In *incremental* model checking [7, 22, 23], verification results for a given LTS are reused to determine property preservation after some alterations. Instead of performing bisimulation checks, they perform additional bookkeeping, which, unlike our approach, still requires at least as much memory as standard model checking. Moreover, we are focussing on property preservation regardless of the behaviour of the input model, whereas they concretely involve the model. Monotonically adding functionality to property formulae is addressed in, e.g., [4]. It could be interesting to see if this is applicable in our setting to update properties.

Finally, verification techniques other than model checking are also being used. For instance, Giese et al. relate input and output models when specifying a transformation by using a theorem prover [9]. This approach is not completely automated, whereas ours is. Also in [21], a theorem prover is used. There, properties are of a structural nature. Instance-based verification of model transformations is described in [13]. They generate a certificate for each model that is transformed. These certificates cannot be used to show property preservation for arbitrary input models.

		ACS	1394-fin	wafer	brdcst (1)	brdcst (2)	c.syst. (1)	c.syst. (2)
verification time (sec.)	*initial model*	1.85	379.08	4.88	3.48	3.48	29.97	29.97
	transformed model	10.23	18,045.13	49.33	83.53	952.85	48,795.28	45,553.27
φ-preservation	*#checks*	3	3	3	7	7	63	63
	result	✓	✓	✓	✗	✓	✗	✓
	time (sec.)	0.01	0.01	0.01	0.616	0.792	1.90	1.90

Table 1: Experimental results

6. CONCLUSIONS

We presented a new technique aimed at verifying the correctness of complex models that result from iterative refinement through model transformation. Experiments show that it vastly outperforms traditional techniques in such cases.

For future research, first, the networks of LTSs concept could be extended to directly support additional features offered by modelling languages such as SLCO. Furthermore, the relation between the formal notion of rule systems and practical languages for the implementation of model transformations needs further study. Finally, property preservation when adding or removing processes could be investigated.

7. REFERENCES

[1] M. Amrani, L. Lucio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. L. Traon, and J. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *Proc. ICST'12*, pages 921–928, 2012.

[2] S. Andova, M. v. d. Brand, and L. Engelen. Reusable and Correct Endogenous Model Transformations. In *Proc. ICMT'12*, 2012.

[3] K. Bartlett, R. Scantlebury, and P. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM*, 12(5):260–261, 1969.

[4] C. Braunstein and E. Encrenaz. CTL-Property Transformations Along an Incremental Design Process. In *Proc. AVOCS'04*, volume 128 of *ENTCS*, pages 263–178. Elsevier, 2004.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[6] B. Combemale, X. Crégut, P.-L. Garoche, and X. Thirioux. Essay On Semantics Definition in MDE - An Instrumented Approach for Model Verification. *Journal of Software*, 4(9):943–958, 2009.

[7] C. Conway, K. Namjoshi, D. Dams, and S. Edwards. Incremental Algorithms for Inter-procedural Analysis of Safety Properties. In *Proc. CAV'05*, volume 3576 of *LNCS*, pages 449–461. Springer, 2005.

[8] L. Engelen and A. Wijs. Checking Property Preservation of Refining Transformations for Model-Driven Development. CS-Report 12-08, Eindhoven University of Technology, 2012.

[9] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. Towards Verified Model Transformations. In *Proc. MoDeVVa'06*, pages 78–93, 2006.

[10] R. v. Glabbeek and W. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[11] J. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. van Weerdenburg, W. Wesselink, T. Willemse, and J. van der Wulp. The MCRL2 Toolset. In *Proc. WASDeTT'08*, 2008.

[12] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques. In *Proc. IFM'10*, volume 6396 of *LNCS*, pages 183–198. Springer, 2010.

[13] G. Karsai and A. Narayanan. On the Correctness of Model Transformations in the Development of Embedded Systems. In *Proc. 13th Monterey Workshop*, volume 4888 of *LNCS*, pages 1–18. Springer, 2007.

[14] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[15] F. Lang. EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-Fly Verification Methods. In *Proc. IFM'05*, volume 3771 of *LNCS*, pages 70–88. Springer, 2005.

[16] R. Mateescu and A. Wijs. Property-Dependent Reductions for the Modal Mu-Calculus. In *Proc. SPIN'11*, volume 6823 of *LNCS*, pages 2–19. Springer, 2011.

[17] T. Mens and P. v. Gorp. A Taxonomy of Model Transformation. In *Proc. GraMoT'05*, volume 152 of *ENTCS*, pages 125–142, 2006.

[18] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[19] A. Narayanan and G. Karsai. Towards Verifying Model Transformations. In *Proc. GT-VMT'06*, volume 211 of *ENTCS*, pages 191–200, 2008.

[20] D. Saha. An Incremental Bisimulation Algorithm. In *Proc. FSTTCS'07*, volume 4855 of *LNCS*, pages 204–215. Springer, 2007.

[21] B. Schätz. Verification of Model Transformations. *Electr. Comm. EASST*, 18, 2009.

[22] O. Sokolsky and S. Smolka. Incremental Model Checking in the Modal Mu-Calculus. In *Proc. CAV'94*, volume 818 of *LNCS*, pages 351–363. Springer, 1994.

[23] G. Swamy. *Incremental Methods for Formal Verification and Logic Synthesis*. PhD thesis, University of California, 1996.

[24] D. Varró and A. Pataricza. Automated Formal Verification of Model Transformations. In *Proc. CSDUML*, pages 63–78, 2003.