The pywmi Framework and Toolbox for Probabilistic Inference using Weighted Model Integration*

Samuel Kolb1, Paolo Morettin2, Pedro Zuidberg Dos Martires1, Francesco Sommavilla2, Andrea Passerini2, Roberto Sebastiani2 and Luc De Raedt1

1KU Leuven
2University of Trento

{samuel.kolb, pedro.zuidbergdosmartires, luc.deraedt}@cs.kuleuven.be,
{paolo.morettin, andrea.passerini, roberto.sebastiani}@unitn.it
francesco.sommavilla@studenti.unitn.it

Abstract

Weighted Model Integration (WMI) is a popular technique for probabilistic inference that extends Weighted Model Counting (WMC) – the standard inference technique for inference in discrete domains – to domains with both discrete and continuous variables. However, existing WMI solvers each have different interfaces and use different formats for representing WMI problems. Therefore, we introduce pywmi (http://pywmi.org), an open source framework and toolbox for probabilistic inference using WMI, to address these shortcomings. Crucially, pywmi fixes a common internal format for WMI problems and introduces a common interface for WMI solvers. To assist users in modeling WMI problems, pywmi introduces modeling languages based on SMT-LIB,v2 or MiniZinc and parsers for both. To assist users in comparing WMI solvers, pywmi includes implementations of several state-of-the-art solvers, a fast approximate WMI solver, and a command-line interface to solve WMI problems. Finally, to assist developers in implementing new solvers, pywmi provides Python implementations of commonly used subroutines.

1 Modeling WMI Problems

In a nutshell, WMI [Belle et al., 2015] traces back to SAT, the problem of deciding if a Boolean formula is satisfiable. #SAT builds on SAT but answers the question of how many models satisfy a formula and WMC extends #SAT by allowing models to be weighted. Like SMT(LRA) extends SAT to answer satisfiability for logical formulas with linear inequalities over real variables, WMI extends WMC to integrate over (instead of summing) over the (possibly infinitely many) weighted models of an SMT(LRA) formula. A WMI problem consists of a support $\phi$, an SMT(LRA) formula that describes all feasible worlds, a weight function $w : \mathbb{R}^n \times \{0,1\}^m \rightarrow \mathbb{R}$ that assigns a weight to every possible world and a set of queries, every query being an SMT(LRA) formula whose probability we want to compute. Worlds are value assignments to the problem variables. Internally, pywmi represents WMI problems as tuples $\langle \text{dom}, \phi, w, Q \rangle$. The domain $\text{dom}$ contains problem variables, their types and, optionally, the valid ranges for numeric variables (e.g., $x \in [0,1]$). Both SMT(LRA) formulas $\langle \phi, q \in Q \rangle$ and weight functions $w$ are represented as Abstract Syntax Trees (ASTs) using the implementation provided by the PySMT library [Gario and Micheli, 2015].

The internal representation is exposed to developers and, for example, probabilistic programming languages can interface with it. For example, a language such as Problog [Dries et al., 2015] produces ground programs to be solved using WMC, an extension with continuous variables can solve ground programs using WMI instead. However, for end-users, pywmi also provides two modeling languages for WMI problems that can be directly parsed to the pywmi representation (see Figure 1 for an overview). Both formats can be used to compactly encode a WMI problem, differing in the syntax used for the expressions (inspired by MiniZinc [Nethercote et al., 2007] or SMT-LIB.v2 [Barrett et al., 2010]).

As an example, consider modelling a factory that produces banana and chocolate flavoured ice cream. Figure 2 shows a toy model of the factory production in the MiniZinc-inspired syntax. Variables $b$ and $c$ represent the amount of banana and...
chocolate ice cream produced in a work day by the factory, while \textit{weekday} is a Boolean variable distinguishing between weekdays and weekends. Intuitively, the production is determined by physical constraints, i.e., being non-negative and not exceeding the capacity of the factory, and market trends causing the preference over flavours to produce to be inverted in the weekends. In this toy example, the density functions combines the probability of weekday vs weekend and a linear combination of the amount of flavours produced, conditioned on \textit{weekday}, that increases with the amount of ice cream produced. The WMI file can also contain evidence, e.g., conditioning on weekend days, and a list of queries to, for example, compute the probability that the banana ice cream production at least doubles the chocolate one (0.40), or the probability of having a particularly high production of any flavour (0.03).

\section{Framework for WMI Solvers}

Given a WMI problem in a standardized format, a WMI solver is employed to calculate weighted model integrals, most frequently, with the aim to compute query probabilities. The core interface for WMI solvers in \textit{pywmi} consists of two methods: 1) computing a weighted model integral, given a domain, support and weight functions; and 2) computing the probability of a set of queries, given a domain, support, weight function and a set of queries. As computing the probabilities of a set of queries can be reduced to a set of weighted model integral computations, \textit{pywmi} offers a default implementation. However, solvers that can only compute query probabilities or offer more efficient ways to compute query probabilities (e.g., using knowledge compilation \cite{Kolb2018}), can override the default implementation.

The solving interface makes it possible to use any supported solver to solve any WMI problem in the common representation format. For users, \textit{pywmi} offers a command-line interface to call or compare different solvers for a problem. Additionally, it can assist with converting file formats, installing new solvers or visualizing WMI problems. For developers, \textit{pywmi} offers a growing amount of common functionality that might be useful to manipulating WMI problems or implementing a new solver, such as, easy-to-use data-structures for WMI problems and domains, wrappers around numeric integration software (e.g., Latte \cite{DeLoera2013}) and symbolic computation systems (e.g., PSI \cite{Gehr2016}) that make it easy to write backend-agnostic solvers, and subroutines for computing with PySMT ASTs, linear inequalities, polynomials, sampling from WMI densities, evaluating points w.r.t. to their feasibility and density, etc.

\section{Comparing WMI Solvers}

As mentioned, \textit{pywmi} allows the performance of different solvers to be compared. Aside from a common interface, however, \textit{pywmi} also provides implementations for many state-of-the-art algorithms. These implementations either wrap around existing tools or libraries, or are (re-)implemented natively in \textit{pywmi}. For solvers with external dependencies, \textit{pywmi} tries to make the installation as easy as possible by automating installations or providing detailed instructions. The supported solvers are: WMI-PA \cite{Morettin2019}, XADD-based path-enumeration \cite{Sanner2011} or bound-resolution (BR) algorithms \cite{Kolb2018}, symbolic SDD-based solver Symbo \cite{Zuidberg2019}, PRAiSE \cite{DeSalvoBraz2016}. Additionally, \textit{pywmi} includes a native implementation of XADDs and the BR algorithm, a fast approximate solver based on rejection-sampling and new WMI solvers are currently being developed using \textit{pywmi}.

Using \textit{pywmi}, we can now easily compare how different solvers behave on different problems. For example, we can compare how two state-of-the-art solvers (WMI-PA and BR) compare on some of the problems they introduce: highly structured xor and mutex problems with sparse inequalities (introduced alongside the BR solver), and synthetic WMI problems with random ASTs as support and weight function (using the WMI-PA problem generator). Running both solvers on these problems shows that the search-based WMI-PA solver and its numeric integration software outperform the BR solver for the WMI problems with random ASTs, while the compilation-based BR solver better exploits the structure of the xor and mutex problems (Figure 3).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{The performance of the solvers varies for different types of problems. For highly structured problems the BR solver performs better, while for less structured problems the PA solver is faster.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{The ice cream production example problem encoded using the MiniZinc-flavored WMI modeling language.}
\end{figure}
References


